Querying a messy web of data with Avalanche

Basca, Cosmin ; Bernstein, Abraham

Abstract: Recent efforts have enabled applications to query the entire Semantic Web. Such approaches are either based on a centralised store or link traversal and URI dereferencing as often used in the case of Linked Open Data. These approaches make additional assumptions about the structure and/or location of data on the Web and are likely to limit the diversity of resulting usages. In this article we propose a technique called Avalanche, designed for querying the Semantic Web without making any prior assumptions about the data location or distribution, schema-alignment, pertinent statistics, data evolution, and accessibility of servers. Specifically, Avalanche finds up-to-date answers to queries over SPARQL endpoints. It first gets on-line statistical information about potential data sources and their data distribution. Then, it plans and executes the query in a concurrent and distributed manner trying to quickly provide first answers. We empirically evaluate Avalanche using the realistic FedBench data-set over 26 servers and investigate its behaviour for varying degrees of instance-level distribution "messiness" using the LUBM synthetic data-set spread over 100 servers. Results show that Avalanche is robust and stable in spite of varying network latency finding first results for 80% of the queries in under 1 second. It also exhibits stability for some classes of queries when instance-level distribution messiness increases. We also illustrate, how Avalanche addresses the other sources of messiness (pertinent data statistics, data evolution and data presence) by design and show its robustness by removing endpoints during query execution.

DOI: https://doi.org/10.1016/j.websem.2014.04.002

Querying a Messy Web of Data with Avalanche

Cosmin Basca[a,*], Abraham Bernstein[a]

[a]*Dynamic and Distributed Information Systems, Department of Informatics, University of Zurich, Switzerland*

Abstract

Recent efforts have enabled applications to query the entire Semantic Web. Such approaches are either based on a centralised store or link traversal and URI dereferencing as often used in the case of Linked Open Data. These approaches make additional assumptions about the structure and/or location of data on the Web and are likely to limit the diversity of resulting usages.

In this article we propose a technique called Avalanche, designed for querying the Semantic Web without making any prior assumptions about the data location or distribution, schema-alignment, pertinent statistics, data evolution, and accessibility of servers. Specifically, Avalanche finds up-to-date answers to queries over SPARQL endpoints. It first gets on-line statistical information about potential data sources and their data distribution. Then, it plans and executes the query in a concurrent and distributed manner trying to quickly provide first answers.

We empirically evaluate Avalanche using the realistic FedBench data-set over 26 servers and investigate its behaviour for varying degrees of instance-level distribution "messiness" using the LUBM synthetic data-set spread over 100 servers. Results show that Avalanche is robust and stable in spite of varying network latency finding first results for 80% of the queries in under 1 second. It also exhibits stability for some classes of queries when instance-level distribution messiness increases. We also illustrate, how Avalanche addresses the other sources of messiness (pertinent data statistics, data evolution and data presence) by design and show its robustness by removing endpoints during query execution.

Keywords: federated SPARQL, RDF distribution messines, query planing, adaptive querying, changing network conditions

1. Introduction

With the advent of the Semantic Web, a Web-of-Data is emerging interlinking ever more machine readable data fragments represented as RDF documents or queryable semantic endpoints. It is in this ecosystem that unexplored avenues for application development are emerging. While some application designs include a Semantic Web data crawler, others rely on services that facilitate access to the Web-of-Data either through the SPARQL protocol or various APIs like the ones exposed by *Sindice*[1] or *Swoogle*[2]. As the mass of data continues to grow—Linked Open Data [5] accounts for 27 billion triples as of January 2011[3]—the scalability factor combined with the Web's uncontrollable nature and its heterogeneity will give rise to a new set of challenges. A question marginally addressed today is how to support the same messiness in querying the Web-of-Data that gave rise to the virtually endless possibilities of using the traditional Web. In other words: *How can we support querying the messy web of data whilst adhering to a minimal, least-constraining set of principles that mimic the ones of the original web and will—hopefully—support the same type of creative flurry?.*

Translating the guiding principles of the Web to the Web-of-Data proposes that we should use a single communications protocol (i.e. HTTP with encoded SPARQL queries) and use a common data

*Corresponding author. Tel: +41 44 635 4318

Email addresses: basca@ifi.uzh.ch (Cosmin Basca), bernstein@ifi.uzh.ch (Abraham Bernstein)

[1] http://swoogle.umbc.edu/

[2] http://sindice.com/

[3] http://www4.wiwiss.fu-berlin.de/lodcloud/state/#domains

representation format (some encoding of RDF), which allows embedding links. In addition, it implicitly proposes that:

(a) we cannot assume any (or control the) distribution of data to servers,
(b) there is no guarantee of a working network,
(c) there is no centralised resource discovery system (even though crawled indices akin to Google in the traditional web may be provided),
(d) the size of RDF data no longer allows us to consider single-machine systems feasible,
(e) data will change without any prior announcement,
(f) there is absolutely no guarantee of RDF-resources adhering to any kind of predefined schema, being correct, or referring/linking to other existing data items—in other words: the Web-of-Data will be a mess and "this is a feature not a bug."

As an example, consider the life sciences domain: here information about drugs, chemical compounds, proteins and other related aspects is published continuously. Some research institutions expose part or all of their data freely as RDF dumps relying on others to index it as in the cases of the CheBi[4] and KEGG[5] datasets, while others host their own endpoints like in the case of the Uniprot dataset.[6] Hence, anybody querying the data will have:

- no control over its distribution, i.e. different copyright and intellectual property policies may prevent access to downloading part or the entire dataset but permit access to it on a per-query basis with potential restrictions like time and/or quota limits,
- no guarantees about the availability and network connectivity of the information sources, i.e. some institutions move repositories or change access policies, resulting in server unavailability,
- no guarantees about content stability as data changes continuously due to scientific breakthroughs/discoveries, and a plethora of schemas are used, i.e. some sub-disciplines may favour dissimilar but overlapping attributes describing their results, have differing habits about using same-named attributes, and use a diversity of taxonomies with varying semantics.

Often-times problem domains and researchers' questions span across several datasets or disciplines that may or may not overlap. Even in the light of this messiness, the data about drugs, chemical compounds, proteins, and their interrelations is queried constantly resulting in a strong need to provide integrated and up-to-date (or current) information.

Several approaches that tackle the problem of querying the entire Web-of-Data have emerged lately, and most adhere to the explicit principles. They do, however, not address the implicit principles. One solution, *uberblic.org*,[7] provides a centralised queryable endpoint for the Semantic Web that caches all data. This approach allows searching for and joining potentially distributed data sources. It does, however, incur the significant problem of ensuring an up-to-date cache and might face crucial scalability hurdles in the future, as the Semantic Web continues to grow. Additionally, it violates a number of the implicit principles locking-in data. Furthermore, as Van Alstyne *et al.* [40] argue, incentive misalignments would lead to data quality problems and, hence, inefficiencies when considering the Web-of-Data as "one big database."

Other approaches base themselves on the guiding principles of Linked Open Data publishing and traverse the LOD cloud in search of the answer. Obviously, such a method produces up-to-date results and can detect data locations only from the URIs of bound entities in the query. Relying on URI structure, however, may cause significant scalability issues when retrieving distributed data sets, since (i) the servers dereferenced in the URI may become overloaded and (ii) it limits the possibilities of rearranging (or moving) the data around by binding the id (*i.e.*, URI) to its storage location. Just consider for example the *slashdot effect*[8] on the traditional web. Finally, traditional database federation techniques have been applied to query the Web-of-Data. One of the main drawbacks with traditional federated approaches stemming from their ex-ante (*i.e.*, before the query execution) reliance on *fine-grained* statistical and schema information meant to enable the mediator to build efficient query execution plans. Whilst these approaches do not assume central control over data, they do assume ex-ante knowledge about it facing robustness hurdles against network failure and changes in the underlying schema and statistics (invalidating implicit

[4] http://www.ebi.ac.uk/chebi/
[5] http://www.genome.jp/kegg/
[6] http://beta.sparql.uniprot.org/
[7] http://platform.uberblic.org/
[8] http://en.wikipedia.org/wiki/Slashdot_effect

principles b and f).

In this paper, we propose AVALANCHE, a novel approach for querying the messy Web-of-Data which (1) *makes no assumptions about data distribution, schema, availability, or partitioning* and is skew resistant for some classes of queries, (2) provides *up-to-date results* from distributed indexed endpoints, (3) is *adaptive* during execution adjusting dynamically to external network changes, (4) *does not require detailed fine-grained ex-ante statistics* with the query engine, and (5) is *flexible* as it makes limited assumptions about the structure of participating triple stores. It does, however, assume that the query will be distributed over triple-stores and not "mere" web-pages publishing RDF. The system, as presented in the following sections, is based on a first prototype described in [3] and brings a number of new extensions and improvements to our previous model.

Consequently, AVALANCHE proposes a novel technique for executing queries over Web-of-Data SPARQL endpoints. The traditional *optimise then execute* paradigm—highly problematic in the Web of Data context in its original conceptualisation—is extended into an exhaustive, concurrent, and dynamically-adaptive meta-optimisation process where fine-grained statistics are requested in a first phase of the query execution. In a second phase continuous query planning is interleaved with the concurrent execution of these plans until sufficient results are found or some other stopping criteria is met. Hence, the main contributions of our approach are:

- a querying approach over the indexed Web-of-Data, without fine-grained prior knowledge about its distribution
- a novel combination of interleaving cost-based planning (with a simple cost-model) with concurrent query plan execution that delivers first results quickly in a setting where join cardinalities are unknown due to lacking ex-ante knowledge
- a reference implementation of the AVALANCHE system

However, despite AVALANCHE's flexible and robust query execution paradigm, the method also comes with a set of limitations discussed in detail in Section 3. The main limitations are as follows:

- AVALANCHE does not benefit from the potential speedup exhibited by intra-plan parallelism since its current computation model does not support UNION-views,
- AVALANCHE can be resource wasteful for some classes of query workloads,
- embracing the WWW's uncertainties (see principles a-f), AVALANCHE neither guarantees result-set completeness nor the same result-set for repeated same-query executions.

Hence, AVALANCHE supports messiness stemming from the lack of ex-ante knowledge at various levels: data-distribution, schema-alignment, prior registration with respect to statistics, constantly evolving data, and unreliable accessibility of servers (either through network or host failure, HTTP 404's, or changes in policy of the publishers).

In the remainder we first review the relevant related work of the current state-of-the-art. The computational model is described in Section 3 while Section 4 provides a detailed description of AVALANCHE. In Section 5 we evaluate AVALANCHE against a baseline system (5.1.1), assess the query planner's quality (5.1.2), observe the system's behaviour when network latency varies (5.1.3) or when endpoints fail (5.1.4) and finally evaluate AVALANCHE with different data distributions (5.2.1) estimating the performance of our system. In Section 6 we present several future directions and optimisations, and conclude in Section 7.

2. Related work

Several solutions for querying the Web-of-Data over distributed SPARQL endpoints have been proposed before. They can be grouped into two streams: **I.** distributed query processing, **II.** RDF indexing, and **III.** statistical information gathering over RDF sources.

Distributed query processing: A broad range of RDF storage and retrieval solutions exist. They can be grouped along the dimensions of *partition restrictiveness* (i.e., the degree to which the system controls the data distribution) and the intended *source addressing space* (i.e., the design goal in terms of physical distribution of hosts from single machine through clusters and the cloud to a global uncontrolled network of servers) as shown

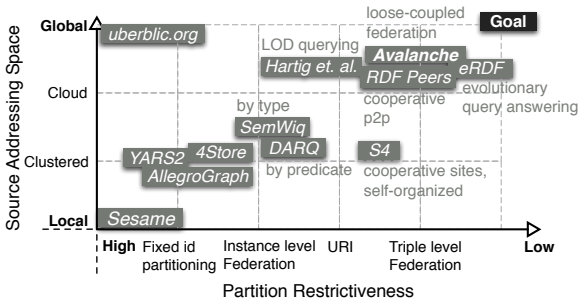


Figure 1: Distributed SPARQL processing systems and algorithms, in relation to the desired goal (high flexibility & global addressing). This figure is not intended to provide an accurate positioning of the systems in the design space.

in Figure 1. Although not intended as a measure of scalability and performance the Figure positions the various approaches relative to the desired goal – a globally addressable and highly flexible system: both paramount features when handling messy semi-structured data at large-scale.

Research on distributed query processing has a long history in the database field [36, 21]. Its traditional concepts are adapted in current approaches to provide integrated access to RDF sources distributed on the Web-of-Data. For instance, *Yars2* [16] is an end-to-end semantic search engine that uses a graph model to interactively answer queries over semi-structured interlinked data, collected from disparate Web sources. Another example is the *DARQ* engine [31], which divides a SPARQL query into several subqueries, forwards them to multiple, distributed query services, finally, integrating the results of the subqueries. Inspired by peer-to-peer systems, *Rdfpeers* [7] is a distributed RDF repository that stores three copies of each triple in a peer-to-peer network, by applying global hash functions to its subject, predicate and object. Stuckenschmidt et. al [37] consider a scenario in which multiple distributed sources contain data in the form of publications. They describe how the *Sesame* RDF repository [6] needs to be extended, by using a special index structure that determines which are the relevant sources to be considered for a query. *Virtuoso* [8]—a data integration software developed by OpenLink Software—is also focused on distributed query processing. The drawback of these solutions is, however, that they assume total control over the data distributions – an unrealistic assumption in the open Web.

Similarly, *SemWIQ* [23] uses a mediator distributing the execution of SPARQL queries transparently. Its main focus is to provide an integration and sharing system for scientific data. Whilst it does not assume fine-grained control over the instance distribution they assume perfect knowledge about their `rdf:type` distribution. Addressing this drawback some [43, 34] propose to extend SPARQL with explicit instructions controlling where to execute certain sub-queries. Unfortunately, this assumes an ex-ante knowledge of the data distribution on part of the query writer. Finally, Hartig et al. [17] describe an approach for executing SPARQL queries over Linked Open Data [5] based on graph search. Whilst they make no assumptions about the openness of the data space, the Linked Open Data rules requires them to place the data on the URI-referenced servers – a limiting assumption for example when caching/copying data. A notable approach to browse the WoD and run structured queries on it is depicted by Sig.ma [38], a system designed to automatically integrate heterogenous web data sources. Suited to handle schema messiness Sig.ma differs from Avalanche mainly in its scope, which is that of aggregating various data sources in the attempt to offer a solution, while Avalanche (tackling data distribution messiness) does not integrate RDF indexes, but "guides" the query execution process to find exact matches.

Other flexible techniques have been proposed, such as the evolutionary query answering system *eRDF* by Guéret et. al [11, 29, 12], where genetic algorithms are used to "learn" how to best execute the SPARQL query. The system learns each time a triple pattern gets executed. As the authors demonstrate, *eRDF* behaves better the more complex the query, while simple queries (one or two triple pattern queries) render low performance. Finally Muehleisen et. al [27] advance the idea of a self organized RDF storage and processing system called *S4*. The approach relies on the principles of swarm-logic and exposes certain similarities with peer-to-peer systems.

RDF indexing: A number of methods and techniques to store and index RDF have been proposed to date, some like Hexastore [41] and RDF3X [28] construct on-disk indexes based on B+Trees while exploiting all possible permutations of *Subjects*, *Predicates* and *Objects* in an RDF triple. Other notable approaches include [2], where RDF is index using a matrix for each triple

term pair – an approach suitable for low selectivity queries, suffering in performance however when highly selective queries are asked. Furthermore GRIN [39] proposes a special graph index which stores "center" vertexes and their neighborhoods leading to lower memory consumptions and faster times to answer graph based queries than traditional approaches such as Jena[9] and Sesame[10].

Query optimization: Research on query optimization for SPARQL includes query rewriting [18], join re-ordering based on selectivity estimations [25, 4, 28], and other statistical information gathering over RDF sources [22, 15]. *RDFStats* [22] is an extensible RDF statistics generator that records how often RDF properties are used and feeds automatically generated histograms to *SemWIQ*. Histograms on the combined values of SPO (*S*ubject *P*redicate *O*bject) triples have proved to be especially useful to provide selectivity estimations for filters [4]. For joins, however, histograms can grow very large and are rarely used in practice. Another approach is to precompute frequent paths (i.e., frequently occurring sequences of S, P or O) in the RDF data graph and keep statistics about the most beneficial ones [25]. It is unclear how this would work in a highly distributed scenario. Finally, Neumann et. al [28] note that for very large datasets (towards billions of triples) as even simple index scans become too expensive, single triple pattern selectivity is not enough to ensure accurate join selectivity estimation. As pattern combinations are more selective, they successfully integrate *holistic sideways information passing* with the recording of detailed join cardinalities of constants joined with the entire graph as means of improving join selectivity. An alternative approach is represented by summarizing indexes as described by Harth et. al. [15] in *data summaries*.

3. Computational Model

Avalanche's computational model diverges from the traditional federated query processing paradigm in several key ways due to the uncertainties of the Web-of-Data (WoD) outlined above. In the following we will detail these characteristics, the assumptions from which they stem and the advantages and disadvantages they introduce while identifying some of the pertinent scenarios that Avalanche is suited for.

Guaranteeing *global completeness*—i.e., a complete result set (or answer set)—on the WoD is impossible due to its uncertainties. Servers may go down (or unreachable) at any given point in time not delivering triples necessary or new servers may appear on the but be unknown to the query engine. However, considering the restricted scope of the endpoints (or sources) selected to participate in a given query we advance the notion of result-set *query-contextual completeness*. By this we refer to the set of all tuples, which constitute the complete query answer if none of the participating endpoints fail.

For these reasons, in Avalanche we focus on optimising for answering SPARQL queries under uncertain conditions and constrains like the FAST FIRST limit modifier used in ORACLE RDB [1]. Consequently, Avalanche is designed to deliver partial results as they become available favouring those that are faster to compose. If the query execution process is not stopped, Avalanche is *eventually complete* in the query-contextual scope. Hence, Avalanche puts more emphasis on the low latency part of the result-set than on completeness by allowing the query requester to specify various uncertain termination conditions (i.e., relative rolling saturation or first answers). In this sense, Avalanche behaves akin to a Web search engine where the first or most relevant results are fetched with the lowest attainable latency while initially ignoring the rest. Thus, Avalanche is suited for exploratory scenarios where the domain is unknown or changes often, situations where bulk data access is limited in some manner (i.e., legal or jurisdictional considerations), or scenarios where at least some results are required fast (i.e., to quickly render the first page with search results from a query).

A distributed query processing system, Avalanche splits the query execution process into three phases as seen in the diagram from Figure 2. The process closely resembles the traditional federated SPARQL processing pipeline: it first identifies the relevant sources to consider, it then retrieves fine-grained statistical information pertinent to the query being executed and finally resolves an optimised version of the original query.

Since finding the optimal plan for a distributed query is NP-hard solutions often rely on heuristics to find plans yielding higher levels of performance

[9] http://jena.sourceforge.net/
[10] http://www.openrdf.org/

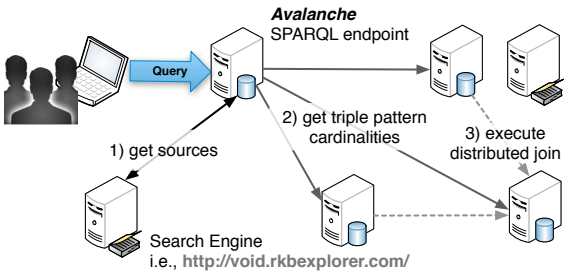


Figure 2: A simplified view of the AVALANCHE execution model illustrating the three major phases: source discovery, statistics gathering, and query planing/distributed execution

[30]. In addition, further complications emerge due to the WoD's underlying uncertainties enumerated before. Hence, AVALANCHE introduces a number of changes to the querying process which depart from the traditional distributed query processing paradigm. In the remainder we discuss its characteristic heuristic and executions strategy.

Heuristics. A heuristic that AVALANCHE employs when exploring the plan composition space is to consider only plans where *any triple pattern of the query can only be answered by one host.* This presents the following main advantages:

1) *generated plans are simpler* and therefore easier to optimise, i.e. using strategies like join-reordering,

2) *generated plans are easier to execute*, i.e., using traditional blocking join / merge physical operators – supported by a wider range of Semantic DBMS's, and

3) *the plan search space is reduced* since all possible plans where a triple pattern is bound to multiple hosts (combinatorial complexity) are not considered when estimating cost.

However, employing this planning heuristic, also introduces the following limitations:

i) *a high number of plans producing empty answer-sets* is generated for queries where the number of participating sites is much larger than the sites where partial results are located (i.e., highly localised queries that make use of widely used terminology),

ii) does not generate plans that contain *unions*.

Avoiding *unions* of partial results can be a severe limitation for some classes of queries while benefitting others. Consider for example the situation were a triple pattern can be answered by more than one host. The selectivity distribution of this triple pattern over selected sites can fall in one of the following situations: the triple pattern can be either *homogeneously selective* i.e., of comparable selectivity on all participating hosts or *heterogeneously selective* i.e., of varying (low and high) selectivity on participating hosts.
The *homogeneously selective* case is simpler since we can consider the union of all pertinent hosts for the given triple pattern. First, by doing so the number of generated plans is reduced by replacing all plans where the triple pattern was bound to one host with one plan that binds the triple pattern to all hosts. Second, the newly generated plan executes faster because it leverages the parallelism of the union operation. Finally the answer-set is larger because all hosts are considered as opposed to only one.
This is not the case when the triple pattern is *heterogeneously selective.* In this situation a union over all sites will severely hinder the performance of executing the plan due to the high latency and high resource utilisation of the high selectivity components of the union. Higher performance can be obtained for a subset of the results by considering only some of the hosts as participating in the union, at the expense of a combinatorial increase in the number of plans to search through.

Execution strategy. AVALANCHE makes use of a concurrent execution strategy of all plans. Doing so confers the following advantages:

1) it has the potential to speed up query execution by leveraging inter-plan parallelism and by warming up local endpoint cache hierarchies, i.e. the same subquery is likely to be requested several times by different concurrent plans - with adequate concurrency control only the first request is executed while all subsequent ones are served from materialised memory views. This of course depends on available memory. For the same reason the execution of multiple *overlapping* queries could be sped up,

2) it attempts to mitigate the negative effect of empty answer-sets since the execution of plans that produce empty result-sets (*unproductive* plans) is intertwined with that of plans that

produce non-empty answers (*productive* plans). Furthermore, *unproductive* plans are in general executed quicker since they can be halted early, when the first empty join is encountered.

Still, this execution strategy can be resource wasteful especially when multiple *non-overlapping* queries are executed. To address this, AVALANCHE makes use of various plan cost model heuristics when estimating plan cost in order to reduce resources wastefulness, essentially aiming to execute those plans deemed *productive* as early as possible. The plan generation process and cost estimation model are detailed in Section 4.

4. The Design and Implementation of an Indexed Web-of-Data Query Processing System

AVALANCHE is part of the larger family of Federated Database Management Systems or FDBMS's [19]. Focusing primarily on answering SPARQL queries over WoD endpoints, AVALANCHE relies on a commonly used data representation format: RDF and SPARQL as the main access operation. In contrast to relational FDBMS, where schema changes are costly and, therefore, happen seldom, the WoD is subjected to constant change, both schema and content-wise. In consequence, the major design contribution of AVALANCHE is that *it assumes the distribution of triples to machines participating in the query evaluation to be unknown prior to query execution.*

To achieve *loose coupling* AVALANCHE adheres to strict principles of transparency as well as heterogeneity, extensibility and openness. When submitting queries to an AVALANCHE endpoint the user does not need to know where data is actually located, ensuring *location transparency.* AVALANCHE endpoints are SPARQL endpoints that can additionally orchestrate the execution of queries according to the model we detail in the following sections. To achieve *replication and fragmentation transparency*, AVALANCHE is also data-distribution agnostic. In addition, participating endpoints are not constrained in any way with regard to the schemas, vocabularies, or ontologies used. Furthermore, over time the federation can evolve unrestrained as new data sources can be added without impacting existing ones.

Akin to peer to peer systems (p2p), AVALANCHE does not assume any centralised control. Any computer on the internet can assume the role of an AVALANCHE-broker. However, AVALANCHE is not a p2p system, since participating sites do not make fractions of their resources—*CPU, RAM*, or *disk*—directly available to other members, nor are they bookkeeping information concerning neighbouring hosts.

Another important distinction to existing federated SPARQL processing systems, lies within the early stages of the query execution. Traditionally, statistical information is indexed *ex-ante*, i.e., ahead of query execution time in the federation's meta-database from where it is later retrieved to aid the source selection and query optimisation processes. AVALANCHE relies on each participating site to manage their respective statistics individually – a trait shared to a varying degree by virtually any optimised RDF-store. Consequently, *query-relevant statistical information is retrieved at the beginning of each query execution phase* as illustrated in Figure 2.

In the following, we will first outline our approach, detailing its basic operators and the actual system using a motivating example. This will lead the way towards thoroughly describing the AVALANCHE components and its novelty.

4.1. System Architecture

The AVALANCHE system consists of the following major components working together in a concurrent asynchronous pipeline: (1) the AVALANCHE *Source Selector* relying on the *endpoints Web Directory* or *Search Engine*, (2) the *Statistics Requester*, (3) the *Plan Generator*, (4) the *Plan Executor Pool*, (5) the *Results Queue* and (6) the *Query Execution Monitor/Stopper* as illustrated in Figure 3.

These components are coordinated into three query execution phases. First, participating endpoints are identified during the *Source Discovery* phase. Second, query specific statistics are retrieved during the *Statistics gathering* phase while finally followed by the*Query Planning and Execution* phase. We will now discuss how all the components are coordinated into these execution phases. The detailed technical description of the elements will be covered in the following subsections.

During *Source Discovery*, participating hosts are identified by the *Source Selector*, which interfaces with a *Search Engine* such as *voID store*,[11]

[11] http://void.rkbexplorer.com/

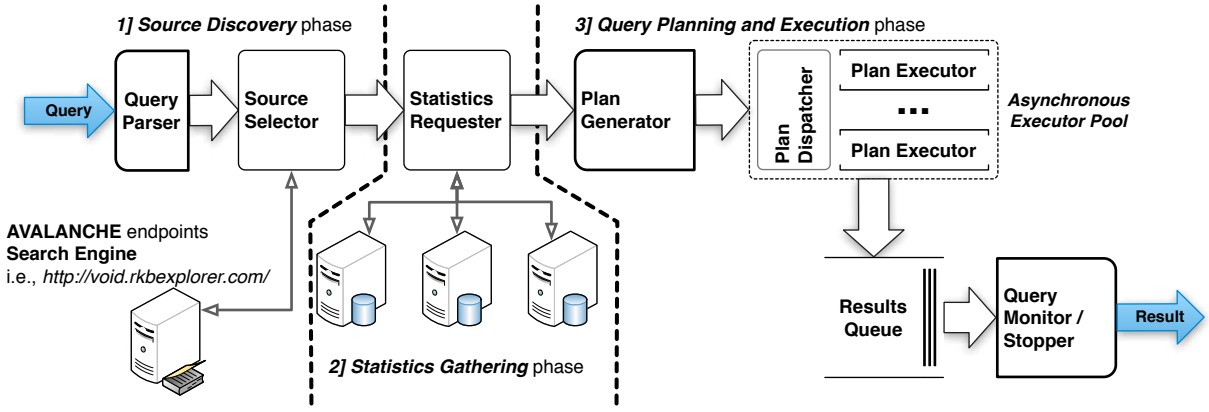


Figure 3: The AVALANCHE execution pipeline

Sindice's[12] SPARQL endpoint, or a *Web Directory*. A lightweight endpoint-schema inverted index can also be used. Ontological prefix (the shorthand notation of the schema, i.e. foaf) and schema invariants (i.e. predicates, concepts, labels, etc) are appropriate candidate entries to index. More complex source selection algorithms and indexes have been proposed [24] that could successfully be used by AVALANCHE given adequate protocol adaptations.

The next step—*Statistics gathering*—queries all selected AVALANCHE endpoints (from the set of known hosts H) for the individual cardinalities $card_{i,j}$ (number of instances) for each triple pattern tp_i from the set of all triple patterns in the query T_Q as detailed in Definition 4.1. The voID[13] vocabulary can be used to describe triple pattern cardinalities when predicates are bound or when schema concepts are used, along with more general purpose dataset statistical information, making use of terms like: void:triples, void:properties, void:Linkset, etc. Additionally, the same can be accomplished by using aggregating SPARQL COUNT-queries for each triple pattern or by simple specialised index lookups in some triple-optimized index structures [41].

Definition 4.1 *Given a query Q, T_Q is the set of all triple patterns $\in Q$ and H the set of all reachable hosts. $\forall tp_i \in T_Q$ and $\forall h_j \in H$, we define $\boldsymbol{card_{i,j} = card(tp_i, h_j)}$ as the **triple pattern cardinality** of triple pattern tp_i on host h_j.*

[12] http://sindice.com/
[13] http://www.w3.org/2001/sw/interest/void/

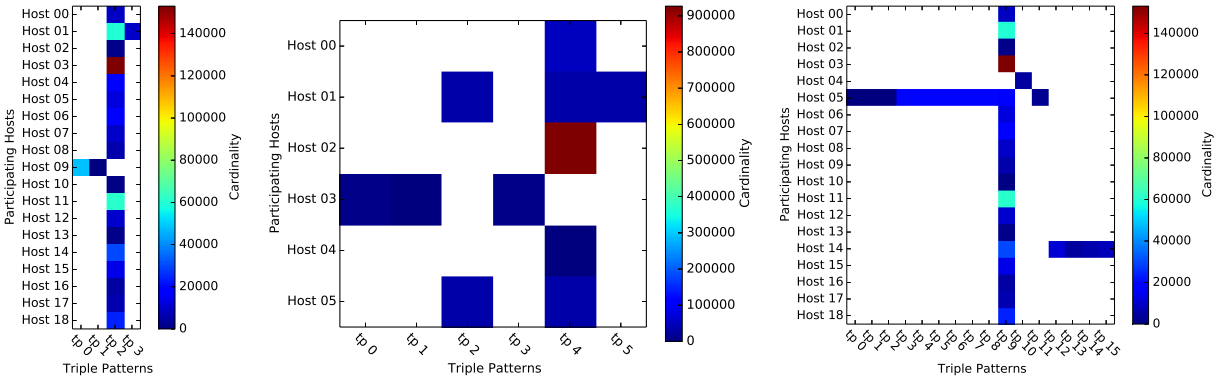
During the *Query Planning and Execution* phase, the *Plan Generator* proceeds with constructing the plan matrix (see Definition 4.2): a two dimensional matrix listing the cardinalities of all triple patterns gathered by the *Statistics Requester* (see Figure 3) of a query by possible hosts. Consider, for example, the plan matrixes for a selection of FedBench queries visualised in Figure 4 as a heat map, where white indicates the absence of triples matching a triple pattern tp_i on some host h_j (i.e., $card_{i,j} = 0$). Focusing on Figure 4 a) we, for example, see that only host-09 has triples matching tp_1.

Definition 4.2 *The matrix PM_Q of size $|H| \times |T_Q|$ defined below is called the **plan matrix**, where the elements $card_{i,j}$ are triple pattern cardinalities as ascertained in Definition 4.1.*

$$PM_Q = \begin{bmatrix} card_{0,0} & \cdots & card_{0,|T_Q|} \\ \vdots & \ddots & \vdots \\ card_{|H|,0} & \cdots & card_{|H|,|T_Q|} \end{bmatrix}$$

The plan matrix is instrumental for the generation of query plans. Every query plan p contains one triple-pattern/host pair (tp_i, h_j) for each triple pattern tp_i in the query T_Q, where all tp_i match at least one triple (i.e., $card(tp_i, h_j) \neq 0$; see Definition 4.3). Thus, planning is equal to exploring the set of possible triple-pattern/host pairs resulting in valid plans. Visually, this corresponds to finding sets of non-zero cardinality squares, where each column is represented exactly once – the assumption that a triple pattern is bound to one host only.

Definition 4.3 *A **query plan** is the set $p = \bigcup(tp_i, h_j)$ that contains exactly one triple-*



(a) $FQ5$ (b) $FQ14$ (c) $FQ33$

Figure 4: Plan matrixes represented as heat-maps for selected Fedbench benchmark queries – for further details about the specific queries and benchmark please refer to Section 5.

pattern/host-pair (tp_i, h_j) *per* $tp_i \in T_Q$, *where* $card(tp_i, h_j) \neq 0$ *and* $h_j \in H$.

While some queries can produce no plans, the universe of all plans (see Definition 4.4) has a theoretical upper-bound equal to $|H|^{|T_Q|}$, however the exact number of plans constructed according to our computational model can be derived using equation 1. Albeit an exponential number of possible plans can theoretically exist, our empirical evaluation suggests that real-world datasets often produce sparse plan matrixes—possibly a consequence of the LoD's heterogeneity—resulting in a significantly lower number of valid plans (i.e., akin to the plan matrixes in Figure 4). Hence, the task of the *Plan Generator* is to explore the space of all possible valid SPARQL 1.1 rewritings of the original query Q by pairing triple patterns from T_Q with available endpoints from H, under the assumption that a triple pattern is bound only to one host. Therefore, unlike traditional DBMS's Avalanche generates *incomplete plans* i.e., where each plan in isolation cannot guarantee result set completeness.

Definition 4.4 *The set of all plans for query* Q, $P_Q = \{p_i \mid p_i$ *is a query plan as in Definition 4.3* $\}$ *is called the* ***query plan space*** *or* ***universe of all plans****.*

$$|P_Q| = \prod_{tp_j \in T_Q} |\{h_i \mid \text{iff } card_{i,j} \neq 0\}|, \qquad (1)$$
$$0 \leq |P_Q| \leq |H|^{|T_Q|}$$

It is important to note that factors such as the sheer size of the Web-of-Data, its unknown distribution, and multi-tenancy aspect may prevent Avalanche from guaranteeing result completeness. Whilst the proposed planning system and algorithm are complete, the execution of all plans to ensure completeness could be prohibitively expensive. Hence, Avalanche will normally not be allowed to exhaust the entire search space—unless the query is simple or the search space is narrow enough. Consequently, Avalanche will try to optimise the query execution to quickly find the **first K** results by first picking plans that are more "promising" in terms of getting results quickly.

As soon as a plan is found, it gets dispatched to be handled by one of the *Plan Executor and Materialiser* workers in the *Executors Pool*. All workers execute concurrently. When a plan finishes, the executor worker places its results, if any, in the *Results Queue*–the queue is continuously monitored by the parallel running *Query Monitor* to determine wether to stop the query execution. Worker slots in the *Executors Pool* are assigned to new workers / plan pairs as soon as plans are generated and slots are available. If the pool is saturated, plans are queued until a worker slot becomes available again. To further reduce the size of the search space, a windowed version of the search algorithm can be employed. Here only the first P partial plans are considered with each exploratory step, thus sacrificing completeness.

In order to optimise execution, Avalanche employs both a common ID space and a set of endpoint capabilities, which we succinctly discuss in the fol-

lowing.

Common IDs. A requirement for executing joins between any two hosts is that they share a common *id* space. The natural identity on the web is given by the URI itself. However some statistical analyses of URIs on the web[14] show that the average length of a URI is 76 characters, while analyses of the Billion Triple Challenge 2010 dataset[15] demonstrate that the maximum length of RDF literals is 65244 unicode characters long with most of the string literals being 10 characters in length. Therefore, using the actual RDF literal constants (URIs or literals) can lead to a high cost when performing distributed joins. To reduce the overhead of using long strings we used a number encoding of the URIs. To avoid central points of failure based on dictionary encoding or similar techniques, we propose the use of a hash function responsible for mapping any RDF string to a common number-based id format. For our experiments, we applied the widely used SHA family of hash functions on the indexed URIs and literals. An added benefit of a common hash function is that the hosts involved in answering a query, can agree on a common mapping function prior to executing the query. Note that this proposition is not a necessary condition for the functioning of AVALANCHE but represents an optimisation that will lead to performance improvements.

Endpoint operations. To optimise SPARQL execution performance AVALANCHE takes advantage of a number of operations that extend the traditional SPARQL endpoint functionality. Whilst we acknowledge that the implementation of these procedures puts a burden on these endpoints their implementation should be trivial for most triple-stores. Some of the operations are either SPARQL 1.1 compliant or can be expressed as plain SPARQL queries, like getting triple pattern cardinalities, total number of triples or executing subqueries which are fully detailed in Appendix A, while others will be internally available in any indexed triple store and "only" need to be exposed (i.e. *set filtering* or *set merge*). From a functional point of view the procedures are classified into two *execution operators* and *state management operators*.

The next subsections will describe the basic AVALANCHE operators and the functionality of its most important elements: the *Plan Generator* and *Plan Executor / Materializer* as well as will explain how the overall execution pipeline stops.

4.2. Query Optimisation

To contextualise AVALANCHE further, consider the example query $Q_{example}$ in Listing 1, executing over the Fedbench[16] benchmark datasets. Specifically the query requests data that are distributed across three life-sciences domain datasets: DrugBank,[17] KEGG,[18] and ChEBI[19]. It is AVALANCHE's goal to find all drugs from DrugBank, together with their URL from KEGG and links to their respective graphical depiction from ChEBI.

Traditionally, query optimisers perform an exhaustive search of the plan universe in order to find the "best" plan given a set of optimisation criteria. The long established *dynamic programming* method is used for this purpose. To further reduce the cost of finding the best plan, the search space is pruned heuristically. A popular heuristic when doing so is to discard all plans with the exception of left-deep ones. Even in the light of these optimisations, exhaustive strategies for traversing the entire plan universe in order to find the best (or lowest cost) plan can become prohibitively expensive for queries where the number of joins is high, i.e. as reported in [32] a number of 15 joins was considered prohibitive circa 2003. Moreover, when dealing with *uncertain constraints* such as FAST FIRST results, RDBMS's like Oracle RDB [1] heuristically execute several plans competitively in parallel for a short interval of time to increase the likelihood of hitting the most relevant cases under the assumption of a ZIPF distribution.

Given that WoD SPARQL endpoints are not under any form of centralised control and network / system failures can occur any time, guarantees about the completeness of a SPARQL query answer cannot be claimed. Consequently, in AVALANCHE we focus on optimising for uncertain constrains akin to the FAST FIRST limit used in Oracle RDB. To this end, AVALANCHE performs an

[14] http://www.supermind.org/blog/740/average-length-of-a-url-part-2

[15] http://gromgull.net/blog/category/semantic-web/billion-triple-challenge/

[16] https://code.google.com/p/fbench/

[17] http://www.drugbank.ca/

[18] http://www.genome.jp/kegg/

[19] http://www.ebi.ac.uk/chebi/

exhaustive search of the plan universe similar to traditional optimisers, with one critical difference: as soon as a plan is generated it is dispatched for execution while the optimiser continues to generate plans. As a first cost-reducing heuristic, we consider only plans where each triple pattern is assigned to one endpoint only. Therefore, each plan is equivalent to a SPARQL 1.1 decomposition of the original query without considering UNION graph patterns. For example one such plan (or decomposition) can be seen in Listing 2, where the SERVICE clause is used to bind triple patterns to endpoints.

```
1  PREFIX rdf: <http://www.w3.org/1999/02/22-↩
       rdf-syntax-ns#>
2  PREFIX drugbank: <http://www4.wiwiss.fu-↩
       berlin.de/drugbank/resource/drugbank/>
3  PREFIX chebi: <http://bio2rdf.org/ns/↩
       bio2rdf#>
4  PREFIX dc: <http://purl.org/dc/elements↩
       /1.1/>
5  SELECT ?drug ?keggUrl ?chebiImage
6  WHERE
7  {
8   ?drug rdf:type drugbank:drugs .
9   ?drug drugbank:keggCompoundId ?keggDrug .
10  ?drug drugbank:genericName ?drugBankName .
11  ?keggDrug chebi:url ?keggUrl .
12  ?chebiDrug dc:title ?drugBankName .
13  ?chebiDrug chebi:image ?chebiImage .
14 }
```

Listing 1: Contextualising example - Life Sciences query from the Fedbench benchmark.

```
1  PREFIX rdf: <http://www.w3.org/1999/02/22-↩
       rdf-syntax-ns#>
2  PREFIX drugbank: <http://www4.wiwiss.fu-↩
       berlin.de/drugbank/resource/drugbank/>
3  PREFIX chebi: <http://bio2rdf.org/ns/↩
       bio2rdf#>
4  PREFIX dc: <http://purl.org/dc/elements↩
       /1.1/>
5  SELECT ?drug ?keggUrl ?chebiImage WHERE {
6   SERVICE <http://drugbank-endpnt/sparql> {
7    ?drug drugbank:genericName ?drugBankName .
8    ?drug drugbank:keggCompoundId ?keggDrug .
9    ?drug rdf:type drugbank:drugs
10  } .
11  SERVICE <http://chebi-endpnt/sparql> {
12   ?chebiDrug chebi:image ?chebiImage .
13   ?chebiDrug dc:title ?drugBankName
14  } .
15  SERVICE <http://kegg-endpnt/sparql> {
16   ?keggDrug chebi:url ?keggUrl
17  }
18 }
```

Listing 2: Motivating example query rewritten as a SPARQL 1.1 federated query.

Plans (or decompositions) can be classified into two categories: *productive plans* – those for which results are found – and *unproductive plans* – those for which no results are found. Considering this, just like in Oracle RDB we adopt the assumption that *the concurrent execution of plans will have a higher probability of yielding results if productive plans are found and dispatched early by the planer.* Hence, Avalanche also executes plans in parallel with the notable difference to Oracle RDB that it sets out to execute all plans until results are found or the stopping criteria are met. As a result the order in which plans are generated is critical, since this is the order in which they are also executed. As our empirical results from Section 5.1.2 show first results are found early during plan generation and execution. For many of the benchmark queries first results also coincide with total query results. A disadvantage of this approach is the apparent wasting of resources. We alleviate this problem by extending the SPARQL endpoint functionality with stateful distributed join processing by caching partial results in memory for the duration of the entire query. In this manner, when the same subquery is part of multiple plans on the same endpoint, the effort of retrieving results from disk is spent only the first time. Furthermore, we assume that expensive and unproductive plans, which would consume resources needlessly, are discarded early by local endpoint optimisers – a feature supported by most industrial-strength RDF stores.

One of the main advantages conferred by this approach is that it relaxes the need for near-exact plan cost estimation. While for traditional query optimisers it is critical to estimate the cost as best as possible because only one plan (the best) is executed, in Avalanche since all plans are executed concurrently the best plans need only be ranked towards the beginning of the execution chain. Hence, the focus falls on the relative ranking of plans to each other. To generate plans efficiently the plan generator has to meet the following criteria:

- it must *generate plans in an order that matches as much as possible the order given by their estimated cost*, with the lowest cost estimate first, and
- *construct plans in an iterative fashion*, since waiting for an exhaustive composition of all plans is expensive – see Definition 4.4 for the upper bound.

Considering these requirements, we created a new graph traversal algorithm which we call: *Priority Queued Greedy DFSs.* The algorithm toggles be-

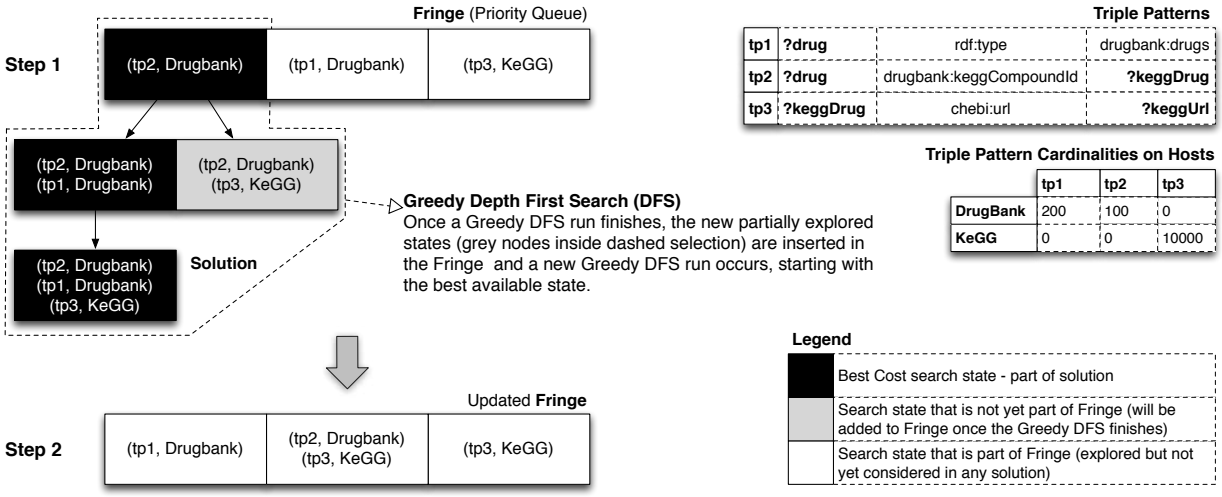


Figure 5: Graphical example of a snapshot of the plan-generator traversal algorithm for a simplified version of $Q_{example}$. For brevity only three triple patterns are considered from $Q_{example}$ while the plan-generator algorithm is detailed over the first step.

tween two modes of operation. First, it starts by seeding the global fringe implemented by a priority queue with all combination of triple pattern - endpoint pairs. Second, a localised *Greedy DFS* is performed starting with the best (or lowest cost) state from the global fringe i.e., node $(tp2, Drugbank)$ in Figure 5. From this point on, expansion is performed using a local fringe, implemented by a stack. Nodes are pushed to the stack in order of their depth and for each depth level in order of their cost estimate. After a solution node is found i.e., ((tp2, Drugbank), (tp1, Drugbank), (tp3, KeGG)), the local fringe is inserted into the global fringe. The local Greedy DFS ensures the second criteria, while the global fringe ensures that multiple DFS searches can be performed efficiently because of the inclusion of partially explored solutions i.e., the grey node $((tp2, Drugbank), (tp3, KeGG))$ in Figure 5. We detail the plan generator algorithm in Section 4.4.

4.3. The Cost Model

Commonly, cost models can be classified into cost models that either aim to reduce the total time to execute the query or strive to reduce the response time or first result(s) latency. The first class of cost models are in general pertinent to single query execution scenarios. Since a complete result set is not in Avalanche's scope the second class of cost functions is desirable. Unlike the comprehensive cost model highlighted by Ozsu and Valduriez in [30] Avalanche features a more relaxed cost model since it does not aim at producing one single cost-optimal plan but instead aims to execute all plans concurrently. Note that in practice concurrency is limited to a number of concurrent operations, a parameter chosen by the administrator (DBA) in line with the desired / possible load of the underlying broker/endpoint hardware. In consequence, since Avalanche needs to rank all generated plans as close as possible to the order of their cost estimates, two simplifying assumptions can be considered:

- *Network*: We assume that network latency and bandwidth are relatively uniformly distributed between participating sites. Although a gross approximation, the assumption holds true in most cases for geographically "near" sites. Furthermore, many participants on the WWW follow this assumption.

- *Distributed Joins*: A widely encountered phenomenon on the WoD, multi-tenancy gives rise to a number of difficulties and problems ranging from management of RDF data to query and index optimisation both locally and at a global scale. Since Avalanche's scope is the indexed WoD, it is unrealistic to assume that full index statistical information is always available or can always be shared between participating sites. Therefore, in the absence of more exact and elab-

orate metrics join selectivity is estimated. The main advantages of this model are: *1*) there is no need for joint distribution statistics to be available and *2*) it bears virtually no computation and network cost. However, there are many fallacies introduced as it offers no guarantees regarding the size of the join between any two BGPs.

In the following we discuss the impact these assumptions have on the cost model.

Selectivity estimation. In the absence of exact statistics (i.e., join cardinalities) regarding *triple patterns* and *basic graph patterns*, selectivity is usually estimated. However, as AVALANCHE starts with the premise that triple pattern cardinalities are know as reported by `getTPCardinality` (Appendix A), triple pattern selectivities are computed and not estimated. For a given triple pattern ***tp*** bound to a given host ***h*** its selectivity represents the probability of selecting a triple that matches from the total number of triples involved and is thus directly computed as follows:

$$sel^h_{tp} = P_{match}(tp, h) = \frac{card(tp, h)}{T_{MAX}} \tag{2}$$

where $T_{MAX} = \sum_{i=0}^{|H|} triples_{h_i}$, with $triples_{h_i}$ representing the total number of triples on host h_i.

Most RDF database management systems (with very few exceptions [28]) *estimate* the selectivity of BGPs. In doing so AVALANCHE discriminates between *star* shaped graph patterns and the rest. Graph theoretic constructs, *star* graph patterns, materialise in the realm of SPARQL queries as groups of triple patterns that join on the same subject or object. For simplicity we will later refer to them as *star graph patterns* or *stars*. Any given basic graph pattern *bgp* can be decomposed into the set of all contained stars referred to as $\mathbb{S}_{bgp}$ and a remainder graph pattern which contains all triple patterns that do not form stars called $\mathbb{NS}_{bgp}$. In consideration of the above, the selectivity of *bgp* is estimated according to the the following formula:

$$SEL^h_{bgp} = \prod_{tp' \in \mathbb{NS}_{bgp}} sel^h_{tp'} \times \prod_{star \in \mathbb{S}_{bgp}} \left(\min_{tp'' \in star} sel^h_{tp''} \right) \tag{3}$$

The equation captures the intuition that non-star pattern triple-patterns are estimated via independent combination of their selectivities. Obviously, independence is not correct but oftentimes found as an acceptable approximation. The selectivity of a star pattern, in contrast, is estimated by the selectivity of its minimal participating triple-pattern.

Cost model. When ranking plans, AVALANCHE employs a common no-preference multiobjective optimization method: the method of *Global Criterion* [42]. AVALANCHE uses this method as an envelope to combine the folowing heuristic objectives:

a) **plan selectivity estimation**: this objective relies primarily on selectivity estimation as it appears in equations 2 and 3 and is defined according to the following equation:

$$SEL_{plan} = \prod_{sq \in SQ_{plan}} SEL^{h_{sq}}_{bgp_{sq}} \tag{4}$$

where *plan* represents a partial or complete plan and SQ_{plan} is the set of subqueries in *plan*.

b) **number of subqueries**: stemming from a *data-locality* assumption (related assertions are usually on the same host) this second heuristic is intended to bias the plan generator towards plans (or partial plans) that will result in query decompositions with fewer subqueries and is defined as follows:

$$SIZE_{plan} = |T_{plan}| - |SQ_{plan}| \tag{5}$$

where *plan* represents a partial or complete plan, $T_{plan} = \{tp_i \mid tp_i \in plan\}$ is the set of triple patterns in *plan*, and SQ_{plan} is the set of subqueries in *plan*.

Since AVALANCHE needs to compare partial plans with various degrees of completion whilst exploring the universe of all plans P_Q the number of subqueries is "normalised" by the number of triple patterns considered so far. Additionally, since the method of global criterion is sensitive to the scaling of the considered objective functions, as recommended in [26], the objectives are normalised into

the uniform [0,1] range. Finally, AVALANCHE minimises the cost of a plan by combining the previous heuristic functions according to the following equation:

$$COST_{plan} = || < SEL_{plan}, SIZE_{plan} > - z^{ideal} || \quad (6)$$

where z^{ideal} represents the ideal or target cost value and the $||.||$ norm is the L_2 norm or the euclidean norm.

One of the main advantages of the cost model defined in this manner, is the flexibility conveyed by the fact that new heuristics can easily be plugged in. Plugging-in an additional element to the cost function would entail extending the cost vector $< SEL_{plan}, SIZE_{plan} >$ with an additional performance indicator as well as z^{ideal} with the desired target value for this indicator. We chose to favour high selectivity plans first over low selectivity ones, mainly due to the assumption that in general they are less costly to execute, thus reducing the time / resource usage penalty in case no results are found. Low selectivity plans are not discarded altogether, but simply given lower priority during execution. Hence, the target value for the first element of the cost function is 0. In addition, the second objective favours plans with fewer distributed joins (fewer subqueries) subscribing to a similar rationale: they are often cheaper to execute by pushing complexity towards local endpoints while avoiding expensive network transfers and connections – a fact particularly detrimental for queries that produce few results. Consequently the target value of the second element of the cost function is also 0 resulting in $z^{ideal} =< 0, 0 >$. Hence, for these these two performance indicators z^{ideal} could be omitted from the formula. This would, however limit the generality of the cost function, as elements with target values other than zero could not be added.

4.4. Plan Generation

As seen in Algorithm 1, the planner will try to optimise the construction of all plans using an informed repeated greedy depth traversal strategy. Due to its repeated nature, plans are not generated in strict ascending order of their estimated cost. Instead they are generated in a partially sorted order primarily dictated by the partial cost estimates from the exploration fringe $\mathbb{F}$. This is achieved by minimising the *cost-estimation* function of each plan $COST_{plan}$, described in Equation 6. As designed, the plan generator's worst case complexity is $O(m^n)$.

Algorithm 1 Plan Generation

Precondition: Q a well-formed SPARQL query, $\mathbb{T}$ the set of all triple patterns $\in Q$

Postcondition: $\mathcal{N}$ a set of search nodes, P a query plan

```
1:  procedure planGenerator(Q)
2:      V ← ∅                                  ▷ V: set of visited nodes
3:      C ← ∅                                  ▷ C: set of closed nodes
4:      F ← nodes(V, T, ∅)                     ▷ F: active exploration fringe
5:      ρ ← 0                                  ▷ ρ: current plan counter
6:      while F ≠ ∅ do
7:          if ρ = MAX_plans then
8:              break
9:          best ← F.pop()                     ▷ best is a leaf search node
10:         if isSolution(best) then
                ▷ emit newly found solution as a plan
11:             emit Plan(Q, best, ρ)
12:             ρ ← ρ + 1
13:             F.sort()                       ▷ sort fringe F based on COST
14:         if best ∉ C then
15:             C ← C ∪{best}
16:             T_nxt ← {tp}, tp ∈ T ∧ tp ∉ triplePatterns(best)
17:             if T_nxt = ∅ then              ▷ T_nxt: next unexplored triple pattern in partial plan
18:                 continue
19:             F ← F ∪ nodes(V, T_nxt, best)  ▷ expand search space

                                               ▷ local fringe expansion function
20: function nodes(V, T, parent)
21:     N ← ∅ ▷ N: the nodes, V: visited queue, T: a set of triple patterns
22:     for tp ∈ T do
23:         for h ∈ H do                       ▷ H: the set of all endpoints
                ▷ create a new search node for tp and h
24:             n ← node(tp, h, parent)
25:             if n ∉ V ∧ n ≠ ∅ then
26:                 V ← V ∪ {n}
27:                 N ← N ∪{n}
28:     N.sort()                               ▷ sort local fringe N based on COST
29:     return N
```

With each exploratory step the size of the global fringe $\mathbb{F}$ increases by the number of sites $|H|$ (line 19). This happens for each expanded state or partial plan represented by a $< tp_i, h_j >$ pair, where $tp_i \in T_Q$ is the current triple pattern and $h_j \in H$ a participating endpoint or host. Not considering

pruning, the algorithm is *complete* and *exhaustive* as it iterates over all possible plans. While traditional optimisers stop and return when the optimal *solution* is found, the planGenerator procedure is not halted and instead each solution or plan is **emitted** to the caller (line 10). The generator procedure is in essence a repeated application of a Greedy Depth First Search algorithm driven by a priority-queue-based fringe, which keeps track of all partial plans explored so far. This ensures that search states (or partial plans) are not visited multiple times. The Greedy DFS aspect is necessary to produce viable plans quickly and is encoded by the partial sort of the local fringe $\mathcal{N}$ in function nodes (line 28). Here the exploration of direct descendant partial plans of the current state is enforced. In contrast, the global fringe $\mathbb{F}$ re-sorts (for efficiency we use a heap) all the partial plans explored so far from all previous Greedy DFS runs (line 13). This is critical since the planner must select for expansion the next best plan available.

Pruning. As the exploration space grows quickly, pruning invalid or $\emptyset$ plans is desired. Early pruning is achieved immediately after the *statistics gathering* phase when the plan matrix PM_Q is available, by removing all hosts (matrix rows) for which the cardinality of all triple patterns is 0. In the absence of triple-patten cardinalities, early pruning would not be possible and the maximum number of plans would have to be considered: $|H|^{|T_Q|}$. Hence, queries that produce a $0_{|H|,|T_Q|}$ plan matrix (zero matrix) are stopped during this early optimisation step.

Furthermore, during execution the same join can be often shared by multiple competing plans. Consequently, joins that are $\emptyset$ (empty) are recorded and used as dynamic feedback for the planner, which then prunes any plan that contains an $\emptyset$ join. This aspect transforms the Avalanche planner into an *adaptive* planner as seen in line 25 of the nodes function.

4.5. Query Execution

As we stated in the previous sections, Avalanche conceptually sets out to execute all plans concurrently. In practice however this can lead to high system load when queries are large (number of triple patterns) and have partial results on many endpoints. In the following we will describe how this problem is addressed in our system. Since any Avalanche endpoint can play both the role of a query broker and a SPARQL endpoint, in order to differentiate between the two roles we will simply refer to the endpoint which orchestrates the distributed execution of the query as *Query Broker* while referring to the rest simply as endpoints. Plans are dispatched for execution given the partially sorted order of their cost estimates. Since Avalanche optimises for FAST FIRST results, fast executing plans are favoured. If no stopping criteria is specified (i.e., LIMIT, timeout, etc) and participating endpoints maintain their availability, Avalanche finds all results every time a query is executed under these conditions, albeit in different orders if no explicit sort is specified. However, since no guarantees can be claimed in a multi-tenant setup like the WoD, due to the unpredictability of external factors, Avalanche looses its deterministic query resolution.

Addressing the Query Broker system load. Once the triple pattern cardinalities are retrieved and the plan matrix PM_Q constructed, the Query Broker is primarily responsible with three tasks, as seen in Figure 3: plan generation, plan execution orchestration and query progress monitoring– to determine when to stop. Except for plan generation, all other tasks are mainly I/O bound. We optimise the plan generation algorithm by making use of *memoization* to store the cost of partially constructed plans while traversing the plan space. The plan execution orchestration process is centered around the *Executors Pool*. Considering its I/O bound nature, an *evented* socket-asynchronous paradigm is a natural fit. Using an event loop driven pool instead of a thread pool when dealing with I/O bound tasks can lead to dramatic improvements in terms of the number of concurrent tasks that can be handled at a fraction of the resources used otherwise. While we cannot directly compare to a thread based pool (i.e., due to implementation impedance mismatches which would result in increased development costs), anecdotal evidence suggests that evented task processors can potentially process several orders of magnitude more tasks than thread based ones, if tasks are non-blocking (e.g., I/O requests). Therefore, we based the implementation of the *Executors Pool* on the popular `libevent`[20] event loop.

Addressing Endpoint system load. While the Query Broker can drive many plans concurrently

[20] http://libevent.org/

due to its asynchronous architecture, the system load of participating query endpoints can still be high. We employ two strategies to reduce this burden on query answering endpoints. First, not all plans are dispatched for concurrent execution at the same time but instead a concurrency limit is set on the *Executors Pool*–similar to the number of worker threads in standard thread-pools, but featuring more workers. Currently, this parameter has to be set manually by the system administrator in concordance to available Query Broker system resources or desired load. Second, each endpoint caches the partial results of each received subquery in memory. Since each plan is executed in order of the selectivity estimation of its composing subqueries, the size of partial results (number of tuples) is kept as low as possible. Clearly, this reduces the cost of executing remote subqueries particularly when the same subquery is requested by multiple plans. This is typically the case when some RDF statements are located on only one site and can be joined with more RDF fragments from other endpoints. In addition, each AVALANCHE endpoint is enhanced with distributed join processing capabilities, also implemented using the same asynchronous evented task processing paradigm.

Plan Execution. As soon as a plan is assigned to a worker, the process described in Algorithm 2 unfolds. Figure 6 illustrates this process for the query $Q_{example}$.

A first step consists of sorting the subqueries (if more than 1) in order of their selectivity estimation SEL^h_{sq} on the designated host h. The distributed join is then executed in left-deep fashion, starting with the most selective subquery, as seen in line 5 and steps 1 and 2 in Figure 6. Necessary for the next phase, the order in which joins occurred is recorded in the J_Q queue. The next phase is optional, since it's an optimisation. When enabled, the partial results that have been produced in the earlier join can be reconciled (filter out the pairs that do not match on the remote site) in reverse order of their counter-part joins (line 6, steps 3,4 in figure). Reconciliation can be naive (send the entire set compressed or not) or optimised. The former is used when the cost of creating the optimized structures is higher than just sending the set. In the latter hashes can be send when the item size is larger than its hash or following [33] *bloom-filters* can be employed. Bloom-filters are space-efficient lossy bit-vector representations of sets by virtue of

Algorithm 2 Plan Execution

Precondition: P a valid query plan, R_Q the AVALANCHE *Results Queue*

```
1:  procedure executePlan(P, R_Q)
2:      r ← ∅                                   ▷ r: the results
3:                                  ▷ P has more than 1 subqueries
4:      if isFederated(P) then
                                    ▷ Sort subqueries in P by SEL^h_sq
5:          SortSubqueries(P)
                                    ▷ distributed join of subqueries
6:          J_Q ← distJoin(P)
                                    ▷ reconcile partial results
7:          distReconciliation(J_Q)
                                    ▷ distributed materialization (||)
8:          S_Q ← distMaterialize(P)
                                    ▷ merge partial results
9:          r ← distMerge(S_Q)
10:     else
11:         r ← sparql(P)           ▷ execute SPARQL query
12:     R_Q ← R_Q ∪ r                           ▷ append results
```

using multiple hash functions for recording each element. Finally results are materialized in parallel (line 7, steps 5,6,7 in figure) and then merged on the host corresponding to the first subquery – the one with the lowest estimated selectivity, (line 8, steps 8,9 in figure). To increase execution performance, since many plans contain the same or overlapping subqueries, a *memoization* strategy is employed. Hence, partial results are kept for the duration of the entire query execution and not just for the current plan. This acts as a site-level cache memory, bypassing the database altogether for "popular" result sets when resources permit.

When the merge is completed, the *Plan Executor* worker process will signal the AVALANCHE *Query monitor* via the *Results Queue*. Note that the finished plans do not contain the final results, as the matches are kept remotely. It is the *Query monitor's* responsibility to retrieve the results and update the overall state of the broker accordingly. In the remainder of this subsection we will describe in detail the inner-workings of the operations described above.

Distributed Join & Reconciliation. The join and reconciliation procedures are detailed in Algorithms 3 and 4 respectively. Joining is implemented in a left-deep fashion while the reconciliation procedure is straight-forward.

One important aspect to note is that the execu-

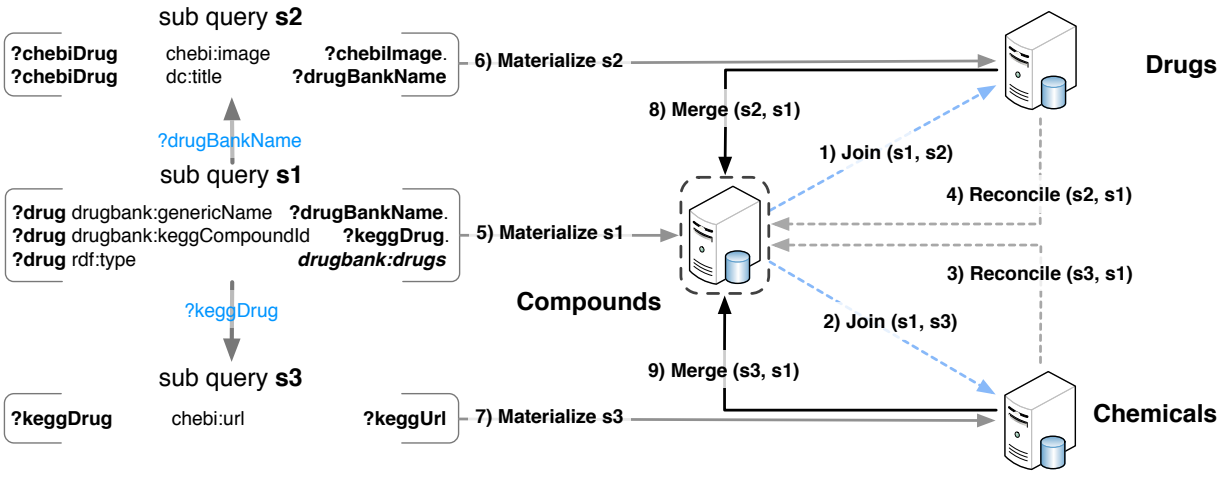


Figure 6: Graphical illustration of the execution process for example query $Q_{ex.}$.

Algorithm 3 Distributed Join

Precondition: P a valid query plan
Postcondition: J_Q a queue, containing the joins in order

```
1:  function distJoin(P)
2:      J_Q ← ∅                          ▷ J_Q: joins queue
                                         ▷ 𝕊: set of all subqueries in P
3:      𝕊 ← subqueries(P)
4:      𝕊.sort()   ▷ sort by selectivity estimation SEL
                   ▷ P has more than 1 subqueries
5:      if isFederated(P) then
6:          while 𝕊 ≠ ∅ do
7:              best ← 𝕊.pop()
8:              for sq ∈ 𝕊 do
9:                  best ⋈ sq                ▷ remote join
10:                 J_Q ← J_Q ∪ {[best, sq]}  ▷ record join
11:     else
            ▷ Execute SPARQL but keep results remotely
12:         sparqlRemote(P)
13:     return J_Q
```

tion of a plan can be stopped (line 6 in Algorithm 4) if the cardinality of a join is 0. This information is recorded and fed back into the planner for dynamic pruning.

Distributed Materialization & Merge. The final execution phases are detailed in Algorithms 5 and 6 respectively. The materialization procedure is executed in parallel on all subquery hosts with the important note that locally kept selectivity es-

Algorithm 4 Distributed Reconciliation

Precondition: J_Q joins queue

```
1:  procedure distReconciliation(J_Q)
2:      J_Q.reverse()
3:      for [left, right] ∈ J_Q do
4:          κ ← reconcile(left, right)
            ▷ stop plan execution when cardinality = 0
5:          if κ = 0 then
6:              halt
```

timations for each subquery in $\mathbb{S}_Q$are updated to actual join cardinalities, available at this stage remotely (line 5 in Algorithm 5). This information is later used to find out the host with the highest partial result cardinality. This host (*best* in line 2 in Algorithm 6) is then used as the "hub" where all other partial results are merged (lines 3-5 in Algorithm 6).[21]

4.6. Stopping the Query Execution

Since we have no control over distribution and availability of the RDF data and SPARQL endpoints, providing a complete answer to the query is an unreasonable assumption except for the cases involving few endpoints and rather simple queries. Instead, the *Query Monitor / Stopper* monitors for the following *stopping conditions*:

[21] For brevity and graphical simplicity of Figure 6, the "Compounds" endpoint (in the middle) was also assigned to be the merge host.

Algorithm 5 Distributed Materialization

Precondition: P a plan
Postcondition: $\mathbb{S}_Q$a queue containing the plan subqueries sorted by cardinality κ

```
1: function distMaterialize(P)
2:     S_Q ← ∅                                  ▷ S_Q: subqueries queue
3:     for sq ∈ P do
           ▷ κ: the cardinality of partial results on sq
4:         κ ← materialize(sq)
5:         S_Q ← S_Q ∪{[κ, sq]}
           ▷ stop plan execution when cardinality = 0
6:         if κ = 0 then
7:             halt
8:     S_Q.sort()                                ▷ sort by κ
9:     return S_Q
```

Algorithm 6 Distributed Merge operation

Precondition: $\mathbb{S}_Q$ subqueries queue
Postcondition: r a valid SPARQL results set

```
1: function distMerge(S_Q)
       ▷ κ: the cardinality of partial results on best
2:     κ, best ← S_Q.popLeft()
3:     while S_Q ≠ ∅ do
4:         sq ← S_Q.popLeft()
                   ▷ merge results from sq on best
5:         merge(best, sq)
               ▷ retrieve the final results from best
6:     r ← getResults(best)
7:     return r
```

→ a global timeout set for the whole query execution,

→ returning the *first* K unique results to the caller,

→ to avoid waiting for the timeout when the number of results is $\ll$ K, we measure relative result-saturation. Specifically, we employ a sliding window to keep track of the last n received result sets. If the standard deviation (σ) of these sets falls below a given threshold, we stop execution. Specifically, we use Chebyshev's inequality: $1 - \frac{1}{\sigma^2}$ [20].

All of the above mentioned stopping conditions can be enabled / disabled independently and in any combination required by a given use-case or desired by the user.

5. Evaluating Avalanche's Robustness Against Messiness

In the introduction we claimed that the Avalanche system provides the capability to query the messy Web-of-Data. Specifically, we claimed that the proposed system: (1) *makes no assumptions about data distribution, schema, availability, or partitioning* and is skew resistant for some classes of queries, (2) provides *up-to-date results* from distributed indexed endpoints, (3) is *adaptive* during execution adjusting dynamically to external network changes, (4) *does not require detailed fine-grained ex-ante statistics* with the query engine, and (5) is *flexible* as it makes limited assumptions about the structure of participating triple stores.

Avalanche is able to provide up-to date results without any ex-ante statistics (2 and 4) by accessing participating triple-stores at run-time and is open due to the limited assumptions it makes on triple-stores (5). Whilst skew resistance (1) and adaptiveness (3) seem possible due to its multi-plan competitive planing/execution strategies (see Sections 4.2 and 4.5) it has not been shown that these strategies are actually successful.

In the following we describe the experimental evaluation of the Avalanche system. Specifically, we will provide empirical evidence ascertaining Avalanche's planner quality and the system's overall robustness to varying data distributions and network conditions such as different latencies and endpoint unreliability. Specifically, we evaluate Avalanche's planer quality as well as robustness against network latency and endpoint stability (in Section 5.1) using a real world dataset. In addition, we show Avalanche's robustness against various data distributions (Section 5.2) using a synthetic dataset.

Experimental setup. For all experiments a cluster of 6 physical machines with 64GB of RAM, 24 AMD Opteron 6174 Cores @2.2 GHz, and running Debian GNU/Linux 6.0.6 64bit was used, connected by a 1 gigabit ethernet switch. In addition the Avalanche broker was executed on a separate machine with 72GB of RAM, 8 Intel(R) Xeon(R) CPU X5570 Cores @2.93GHz, and running Fedora release 12 (Constantine) 64bit. For all evaluations the following stopping conditions were considered unless specified otherwise:

→ a global *timeout* of 300 seconds (5 minutes),

→ *first K* unique results set to 1000 and
→ *relative-saturation* of 90%.
Additionally, the concurrency limit was set to 128 concurrently executing plans.

5.1. Evaluation Setting I: Analyzing Avalanche with real-world data

To evaluate the generalizability of our results to a real-world setting we chose a real-world dataset specifically tailored for the evaluation of federated RDF stores. This subsection first outlines the dataset, its distribution to hosts, the queries used and then discusses Avalanche's execution results on this dataset.

The Data and its Distribution. We chose the recently published Fedbench[22] [35] dataset as it comes pre-partitioned using a real-world partitioning schema and, additionally, offers 36 SPARQL queries. For summarized statistics about each participating dataset refer to Table 1.

Following the natural partitioning of the benchmark we adopted the assumption that each dataset is published on its own distinct server. For bigger datasets such as Geonames and DBPedia we assumed in addition that the publishers decided to further split the data into multiple RDF stores. We captured this by splitting some of the larger datasets as detailed in Table 2. Hence, additional distribution messiness was introduced by splitting the Geonames triples randomly over 11 hosts while for DBpedia larger dumps were distributed to single hosts and the smaller ones were integrated into the *Other* Avalanche endpoint.

The Queries. The triple store[23] we used for implementing Avalanche endpoints does not currently support SPARQL features beyond traditional BGP pattern matching. Hence, we ignored all Fedbench queries that contain the OPTIONAL and FILTER graph pattern modifiers. This is a limitation of the current system and evaluation, which we discuss in detail in Section 6. Additionally, as UNION graph patterns are not supported either, queries containing the operator were split and executed as separate queries, which is aligned with the common practice of executing unions as individual subqueries in parallel. We supplemented the resulting 33 Fedbench queries with another 5 more complex queries from the life sciences domain, as listed in Appendix D. The translation table to the original names (where applicable) is available in Appendix B.

5.1.1. Experiment #1: Avalanche vs. a Baseline System

In this first experiment we intend to better understand through empirical evidence, *the performance gains (or potential shortcomings) that the computational model embraced by* Avalanche *introduces*. Hence, we implemented a *baseline* prototype where the core idea of concurrently executing multiple *simpler* decompositions of the original query is dropped. In contrast to Avalanche the query answer-set is constructed by:

- keeping relevant state (i.e., partial results) in a local repository and
- executing a single optimal query plan generated akin to traditional query optimisation techniques.

Although there are multiple possible execution models that could be considered baselines, one approach is to first multicast query Q to all participating sites. Second, each site would remove triple patterns for which it has no match from Q and return the matching triples. Third, Q would be run against a local repository of the triples returned from all participating hosts. The decision to discard triple patterns—in effect mapping $Q \mapsto Q_{known}$, where Q_{known} is the part of the query known to the server—is carried out by each participating endpoint individually and is implemented as defined in Equation 7:

$$T_{Q_{known},h} = \{tp_i \mid \forall tp_i \in T_{Q,h}, \text{ iff } card(tp_i, h) > 0\} \quad (7)$$

where $T_{Q_{known},h}$ represents the "known" set of triple patterns composing query Q_{known} on the current host h. Other triple pattern exclusion rules can be imagined, i.e. discard all triple patterns if the predicate belongs to an unknown namespace – provided namespace information is available. After all or some of the partial results are retrieved from the remote SPARQL endpoints, they are stored in a local RDF store. Since in the case of

[22] http://code.google.com/p/fbench

[23] An in-house and update-able extension of *Hexastore* was used as the RDF store technology behind all Avalanche endpoints in our evaluations.

Table 1: Fedbench datasets statistics

Collection	Dataset	version	# triples	Dataset	version	# triples
Cross Domain	DBpedia subset	3.5.1	43.6M	Jamendo	2010-11-25	1.05M
	NY Times	2010-01-13	335k	GeoNames	2010-10-06	108M
	LinkedMDB	2010-01-19	6.15M	SW Dog Food	2010-11-25	104k
Life Sciences	DBpedia subset	3.5.1	43.6M	Drugbank	2010-11-25	767k
	KEGG	2010-11-25	1.09M	ChEBI	2010-11-25	7.33M
SP^2Bench	SP^2Bench 10M	v1.01	10M			

[a] Data available from http://code.google.com/p/fbench/wiki/Datasets

Table 2: The distribution of the Fedbench dataset to Avalanche hosts

Dataset	Avalanche Host	#triples	Dataset	Avalanche Host	#triples
NY Times	News	314k	LinkedMDB	Movies	6.14M
Jamendo	Music	1.04M	SW Dog Food	SW	84k
KEGG	Chemicals	10.9M	ChEBI	Compounds	4.77M
Drugbank	Drugs	517k	SP2B-10M	Bibliographic	10M
Geonames	Geography_1	9.98M		Geography_7	9.95M
	Geography_2	9.99M		Geography_8	9.99M
	Geography_3	9.93M		Geography_9	9.99M
	Geography_4	9.94M		Geography_10	9.99M
	Geography_5	9.98M		Geography_11	7.98M
	Geography_6	9.98M			
DBPedia subset	Infobox_Types	5.49M		Infobox_Properties	10.80M
	Titles	7.33M		Articles_Categories	10.91M
	Images	3.88M		SKOS_Categories	2.24M
	Other	2.45M			

SPARQL SELECT queries the answer-sets R_i are tables where columns correspond to projection variables and therefore not graphs G_i as would be the case of SPARQL CONSTRUCT queries, a translation process from tuples to triples needs to be implemented. This is a necessary step as to reconstruct locally the subgraph $G_{known} = \bigcup G_i$. A solution would be to transform each of the Q_{known} SELECT queries to equivalent CONSTRUCT queries. Finally, the engine is left with the task of re-executing the original query Q on the local graph G_{known}.

Limitations of the Baseline System. While conceptually simpler, a number of hurdles render the implementation non-trivial. First, it is possible that some of the reduced queries Q_{known} may not contain any selective triple patterns from Q because the respective hosts do not "understand" those patterns. In the worst case the reduced $Q_{known} \equiv < s, p, o >$ which would trigger the requester to retrieve the entire remote knowledge-base. Second, since the final results for Q can only be computed after obtaining G_{known} two execution strategies emerge:

i) Wait until all G_i partial graphs are retrieved and then execute Q on G_{known}. This is suitable for cases where the partial graphs are inexpensively obtained and/or the query is complex.

ii) Build the final result-set incrementally by executing Q every time a partial graph G_i is merged with the local G_{known} repository. This strategy obviously pays off when (some) partial triples sets are expensive to obtain additionally offering the possibility of an early stop when Q is satisfied without having to wait for all partial graphs. However, it incurs the cost of executing Q with each retrieved partial set of RDF triples i.e., returned by each site.

```
1 PREFIX ex: <http://example.org/>
2 SELECT * WHERE {
3     ?x ex:p1 ?y .
4     ?y ex:p2 ?z .
5     ?z ex:p3 ?u .
6 }
```

Listing 3: Example query Q'_{ex}.

Finally, the method is not complete since it is possible that $\bigcup G_i \subset G_{needed}$, where G_{needed} is the minimal set of triples needed to construct the complete result set for Q. For example consider

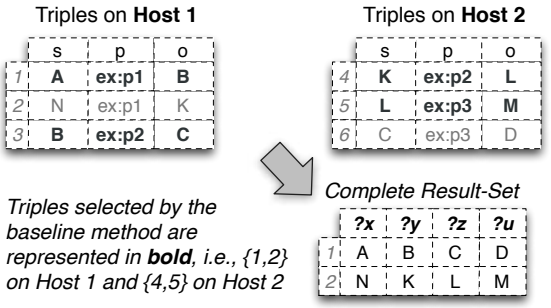


Figure 7: Triples distribution for two hypothetical sites with the complete result-set for query $Q^{'}_{ex}$.

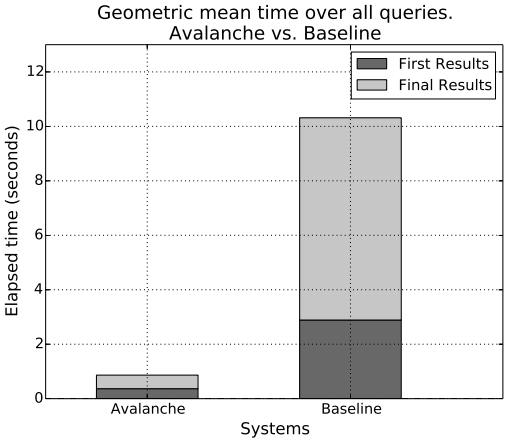


Figure 10: Geometric mean of the execution time over all queries: Avalanche vs. the Baseline System.

the case illustrated in Figure 7 where query $Q^{'}_{ex}$ (from Listing 3) executes over two sites. By this strategy $Q^{'}_{ex}$ produces no results even though the complete result-set contains two tuples. In contrast Avalanche is (eventually) complete since it considers all possible decompositions of Q and not just some decompositions like Q_{known}.

Results. Based on the assumption that the selectivity distribution of the generated Q_{known} subqueries on participating endpoints is Zipf-ian, we chose to implement the pipelined execution model due to its obvious performance benefits. Furthermore, the same asynchronous execution paradigm as in Avalanche was used in the baseline, while G_{known} was implemented by a fast in memory indexed RDF store[24]. A consequence of this choice is that the same stopping conditions that Avalanche employs can be used to determine wether the engine should stop the query execution or not, hence, eliminating other unknown hidden factors when comparing the two systems.

The time taken to complete all the considered Fedbench queries by both systems is graphed in Figure 8. With very few exceptions Avalanche proved to be faster than the *baseline* system. When retrieving first results the baseline system is slower than Avalanche in 65% of the queries, becoming slower for 92% of the queries by the time final results are retrieved. This is better captured in Figure 10, where the geometric mean over all queries is computed. Clearly, for the 38 selected Fedbench queries Avalanche exhibits superior average performance for both cases: retrieving first results and achieving query completion.

Furthermore, as mentioned previously the baseline system is not guaranteed to be complete, a fact exhibited by queries: $FQ11$, $FQ14$, $FQ21$, $FQ23$, $FQ25$, $FQ26$, $FQ28$ and $FQ33$ as seen in Figure 9, which depicts the recall for all queries. In contrast, Avalanche exhibits full recall for most queries with the exception of queries: $FQ8$, $FQ11$, $FQ28$, $FQ30$, $FQ31$, $FQ34$ and $FQ37$ under a time-out of 5 minutes (the same was set for the baseline). The *ground-truth* —total number of results— used to compute the recall was obtained by running all Avalanche plans exhaustively acquiring thus all possible results for each query. This was achieved by disabling all the stopping conditions: timeout, first-k results and relative saturation.

The baseline system although slower for most benchmark queries and incomplete for some, exhibits some positive properties. First, it is of a much more simple design than Avalanche and finally for some classes of queries it can be faster than Avalanche. For example for query $FQ7$ the baseline system completes with 4.6 seconds faster than Avalanche while for query $FQ30$ first results are retrieved marginally (0.37 seconds) faster than Avalanche. As stated above one of the main design limitations of the baseline is represented by the fact that completeness cannot be guaranteed. Even though we implemented the baseline using the same concurrent asynchronous query execution paradigm as in Avalanche a number of potential bottlenecks still exist. A first limiting factor is the

[24]We used the IOMemory RDF store provided by the `rdflib` package: https://github.com/RDFLib

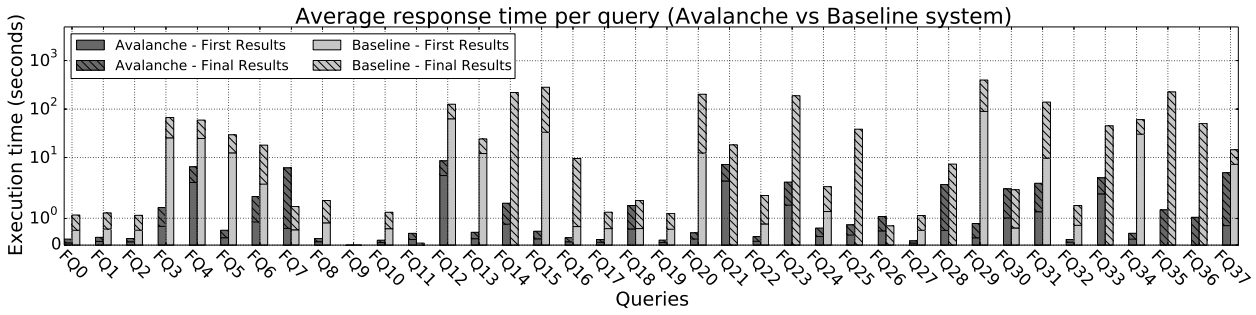


Figure 8: Average query execution times for each of the Fedbench queries. AVALANCHE vs. the Baseline System.

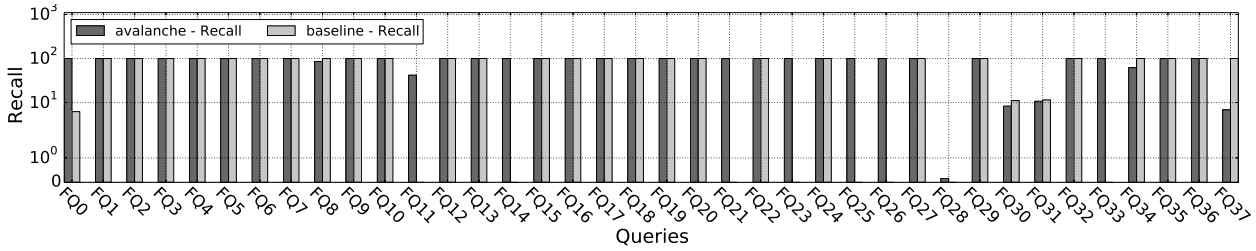


Figure 9: Recall for each of the Fedbench queries. AVALANCHE vs. the Baseline System.

Table 3: Statistical information and query runtime breakdown for the Baseline system on all queries

query	num. query runs[a]	avg. load time (s)[b]	avg. query time (s)[c]	total triples recv.[d]	query	num. query runs[a]	avg. load time (s)[b]	avg. query time (s)[c]	total triples recv.[d]
FQ0	3	0.0003	0.1748	1	*FQ19*	1	0.0123	0.5226	249
FQ1	1	0.0065	0.5123	25	*FQ20*	6	18.0261	0.0934	947537
FQ2	2	0.0004	0.2766	1	*FQ21*	8	1.8014	0.0698	153450
FQ3	18	0.9735	1.0130	206211	*FQ22*	2	0.1137	0.3232	2430
FQ4	17	1.5448	0.0372	385055	*FQ23*	15	7.4315	0.0527	976104
FQ5	15	0.8346	0.0414	132931	*FQ24*	3	0.1865	0.1760	6842
FQ6	6	1.4098	0.9325	79857	*FQ25*	18	1.6552	0.0344	284896
FQ7	3	0.0961	0.1853	2810	*FQ26*	1	0.0870	0.5571	1139
FQ8	1	0.1052	0.5886	1158	*FQ27*	1	0.0007	0.5519	2
FQ9	0	-	-	-	*FQ28*	2	2.3257	0.2940	44087
FQ10	1	0.0303	0.5261	318	*FQ29*	5	34.8218	2.7780	1705932
FQ11	0	-	-	-	*FQ30*	2	0.3084	0.3861	9472
FQ12	19	2.3080	0.3670	595434	*FQ31*	2	26.7686	13.4666	776692
FQ13	2	3.3561	0.2621	138132	*FQ32*	1	0.0517	0.5352	655
FQ14	4	28.7477	0.2759	983324	*FQ33*	18	1.5853	0.4695	288054
FQ15	6	21.8694	11.1893	1114704	*FQ34*	1	0.7198	28.2120	19367
FQ16	3	1.9255	0.1989	54619	*FQ35*	5	24.8012	0.1232	1114611
FQ17	1	0.0302	0.5258	554	*FQ36*	18	1.7777	0.4739	386434
FQ18	3	0.1056	0.2032	3816	*FQ37*	1	3.2252	1.3490	87599

[a] the input query is run repeatedly every time new triples are received
[b] average time – in seconds – to load the newly received triples into the local RDF store
[c] average time – in seconds – taken for each input query run
[d] total number of triples transferred over the network from all endpoints

way in which the query is being executed: by fetching all pertinent (according to Equation 7) triples locally. Intuitively, at least for more demanding classes of queries (i.e., with more joins, or complex shapes), this can easily lead to a large portion of triples to be identified as "pertinent" for the given query and therefore transferred locally. Looking at Table 3 we can clearly observe that for 50% of the benchmark queries, the baseline retrieves anywhere between 100'000 to 1'700'000 triples, while for very few queries the number of triples retrieved counts in the hundreds. Clearly, this represents a bottleneck since not all triples are received at the same time, and in some cases those triples that contribute to the final result are found later in the execution of the given query. Another potential bottleneck is represented by the local RDF store we employed. We opted for an in-memory indexed store to diminish the performance penalties introduced by the loading of new triples as they arrive and at the same time offer high performance for most queries. As can be observed in Table 3 most queries are answered on average below 1 second, however for some queries (e.g., $FQ15$, $FQ29$, $FQ31$ and $FQ34$) the time to rerun the original query on local data is on average quite high ranging from ca. 3 seconds to ca. 30 seconds. This could be explained by the set-based join algorithm used (more expensive than sorted merge-join) since the RDF store does not keep sorted indexes (but dictionary based) to aid the loading / indexing process at the expense of slower execution times for more complex queries.

In light of these results, we can safely say that Avalanche exhibits significant performance and conceptual benefits over the naive baseline system.

5.1.2. Experiment #2: Planner Quality Assessment

In this second experiment we intend to analyse the *quality* of the planning algorithm and cost model that Avalanche uses. Consequently, we:

- compare the performance exhibited by Avalanche with that of a similar system driven by an *oracle planner* and,
- observe the relative ranking of productive plans within the query plan universe P_Q as generated by the Avalanche plan generator.

Comparison to an Oracle Planner. In order to observe to what extent the asynchronous concurrent execution of plans improves the overall performance of query answering in Avalanche we constructed an *oracle planner* (see Definition 5.1).

Definition 5.1 *An **oracle planner** is a plan generator connected to an oracle, akin to an **oracle machine**, i.e. a Turing machine connected to an oracle.*

A drop-in replacement for the Avalanche *Plan Generator*, the oracle planner has perfect knowledge about which of the Avalanche generated plans are productive (i.e., have results) and which are not (i.e., do not find any results). To obtain the productive plans for each query, we serialised the plans for which results were found while running Avalanche without stopping conditions, to disk. We then order these plans according to the same order as Avalanche. Consequently, the oracle planer only generates the plans for which results are found without the time penalty incurred by the exhaustive plan space traversal of the cost-based *Plan Generator*.

It is important to note that for most queries with the exception of $FQ0$, $FQ7$ and $FQ31$ (see Table 4) there is only a single plan which is productive and therefore the *oracle planner* is in this cases equivalent with an omniscient planner where the optimal query decomposition is found.

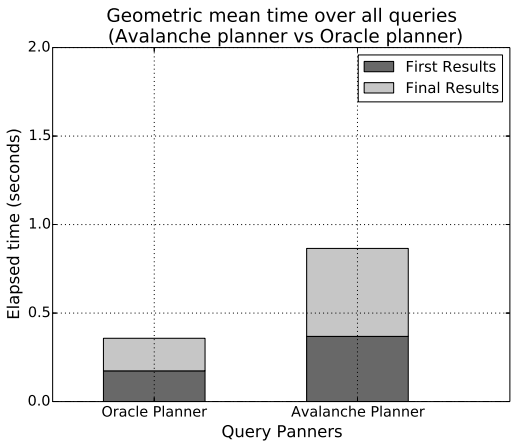


Figure 11: Geometric mean of the execution time over all queries: Oracle vs. Avalanche Planner.

For the experiment we ran all benchmark queries with the oracle planner and compared the performance of query execution to the Avalanche cost model based planner. The number of productive plans for all of the benchmark queries is reported

Table 4: Total possible plans and first productive plan rank as generated by AVALANCHE

query	*FQ0*	*FQ1*	*FQ2*	*FQ3*	*FQ4*	*FQ5*	*FQ6*	*FQ7*	*FQ8*	*FQ9*	*FQ10*
max plans[b]	26^1	26^2	26^3	26^5	26^5	26^4	26^4	26^4	26^1	26^1	26^1
# plans[c]	6	26	1	18	324	18	180	2592	1	0	1
# productive plans[d]	6	1	1	1	1	1	1	2	1	0	1
1^{st} **plan**	**1**	**25**	**1**	**1**	**2**	**3**	**23**	**48**	**1**	-[a]	**1**
query	*FQ11*	*FQ12*	*FQ13*	*FQ14*	*FQ15*	*FQ16*	*FQ17*	*FQ18*	*FQ19*	*FQ20*	*FQ21*
max plans[b]	26^2	26^5	26^7	26^6	5^{26}	26^3	26^3	26^4	26^5	26^3	26^5
# plans[c]	26	18	1	10	10	7	1	126	1	5	594
# productive plans[d]	1	1	1	1	1	1	1	1	1	1	1
1^{st} **plan**	**2**	**6**	**1**	**6**	**2**	**7**	**1**	**20**	**1**	**1**	**71**
query	*FQ22*	*FQ23*	*FQ24*	*FQ25*	*FQ26*	*FQ27*	*FQ28*	*FQ29*	*FQ30*	*FQ31*	*FQ32*
max plans[b]	26^2	26^5	26^3	26^3	26^5	26^3	26^9	26^5	26^2	26^2	26^1
# plans[c]	24	180	1	18	49	1	270	45	104	104	1
# productive plans[d]	1	1	1	1	1	1	1	1	1	3	1
1^{st} **plan**	**9**	**1**	**1**	**2**	**1**	**1**	**1**	**1**	**27**	**20**	**1**
query	*FQ33*	*FQ34*	*FQ35*	*FQ36*	*FQ37*						
max plans[b]	26^{16}	26^{12}	26^{16}	26^{11}	26^6						
# plans[c]	18	1	10	18	324						
# productive plans[d]	1	1	0	0	1						
1^{st} **plan**	**6**	**1**	-[a]	-[a]	**17**						

[a] query has no results
[b] maximum number of plans if no triple-pattern cardinalities are available ≡ upper bound
[c] maximum number of possible plans deduced when triple pattern cardinalities are considered
[d] total number of plans (from all possible plans) for which results are found

in Table 4. As can be seen, all queries feature 1 productive plan (or 0 if query has no results) with the exception of queries $FQ0, FQ7$ and $FQ31$ which produce 6, 2 respectively 3 productive plans.

The results of running all 38 Fedbench queries comparing the standard AVALANCHE planner with the *oracle planner* are depicted in Figure 12, while the geometric mean over all queries when comparing the execution times yielded by the two planners is shown in Figure 11. While for 25 of the queries the absolute elapsed time (wall-clock time) difference is negligible as seen in Figure 12, for queries $FQ1$, $FQ4$, $FQ6-7$, $FQ18$, $FQ21$, $FQ23$, $FQ28-31$, $FQ33$ and $FQ37$ AVALANCHE was between $\approx$ 2 to $\approx$ 33 times slower than the oracle approach. However, looking at Figure 11, AVALANCHE was $\approx$ 2.5 times slower in the geometric mean than the oracle driven system over all benchmark queries.

In general this difference is to be expected. The effort of discarding (and executing) unproductive plans in conjunction with the plan space exploration takes time. Hence, the AVALANCHE planner is naturally slower than a no-effort planner (like the oracle planner). However, as exhibited by Figure 12 the delays are clearly limited and acceptable to many applications. Hence, AVALANCHE exhibits a good performance in the conditions of this evaluation when acting solely on join-estimate heuristics.

Plan ranking. As can be seen in absolute values in Table 4 and normalised relative to total number of plans in Figure 13 AVALANCHE succeeds in assigning a low rank (1 ≡ best rank) to the first productive plan. When the number of possible plans is large, the simple selectivity-estimation-based cost model will assign higher ranks, as is the case of query $FQ21$ where the first productive plan is the 71^{st} plan generated out of 594 possibilities. However, due to the asynchronous-concurrent manner in which plans are executed, the negative effect of assigning higher ranks to plans (the rank is equivalent to the plan's generation order) is mitigated to a relatively high degree as shown in the previous analysis agains the oracle planner, i.e. non-productive plans are quickly discarded after the first empty join.

5.1.3. Experiment #3: Varying Network Latency

Changing network conditions can impede the execution of any distributed SPARQL processing. Two critical network factors stand out: bandwidth and latency. Since the slowdown effect of a low-bandwidth connection can in general be overcome with a certain degree of success by either compressing the message or making use of binary communication protocols and since AVALANCHE employs *bloom filter* optimized joins to reduce communication I/O, we decided to focus our attention in this

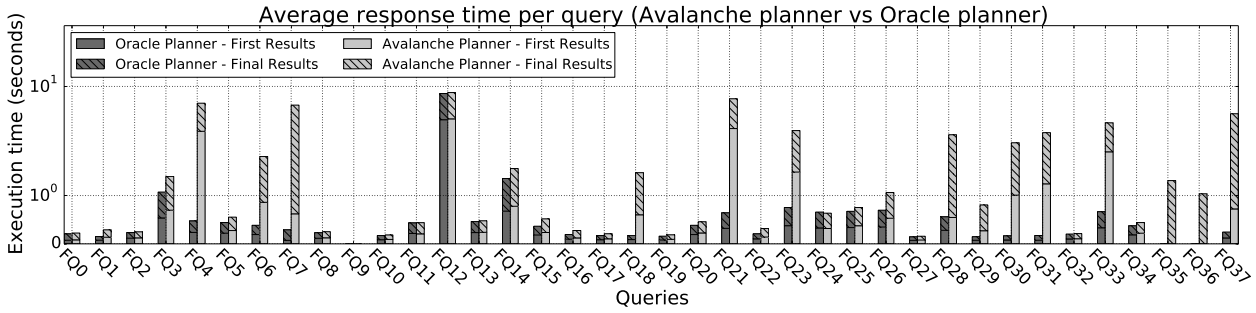


Figure 12: Average query execution times for each of the Fedbench queries. AVALANCHE planner vs. Oracle planner.

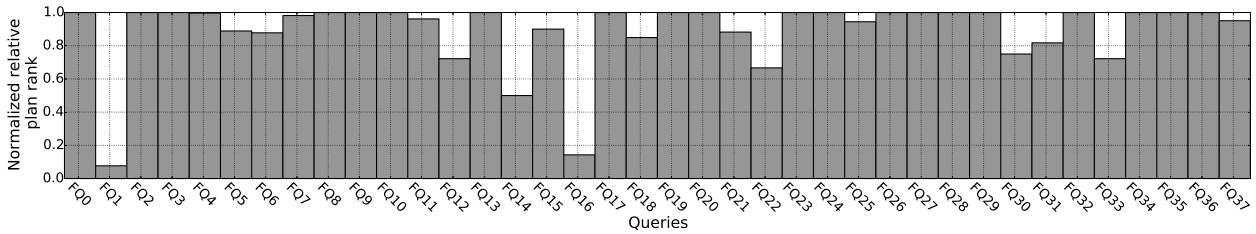


Figure 13: Normalised relative plan ranking: first plan compared to the possible number of plans / query for each Fedbench queries.The higher the bar the better, i.e. productive plans get executed sooner.

experiment on connection latency. The majority of requests in the AVALANCHE system are between the AVALANCHE broker and the participating endpoints. Hence, for this experiment the connection between the broker and each endpoint was routed through a TCP delayer proxy, which would introduce delays according to a predefined configuration. We chose to simulate three types of latency distributions:

- *No Delay* → a *local cluster* network with negligible (close to 0 s) connection latency,
- *Gamma 1* → a *fast network* with an average connection latency of 0.3 seconds. Simulated by a gamma distribution with $\alpha = 1$ & $\beta = 0.3$ (Figure 14),
- *Gamma 2* → a *slow network* with an average connection latency of 3 seconds. Simulated by a gamma distribution with $\alpha = 3$ & $\beta = 1.0$ (Figure 14).

Additionally, the TCP socket buffer size was set to the standard value of 16KB.

AVALANCHE successfully finds results for all the considered benchmark queries under all simulated latency variations. Looking at Figure 15 we can clearly observe that the speed with which AVALANCHE answers queries across the different

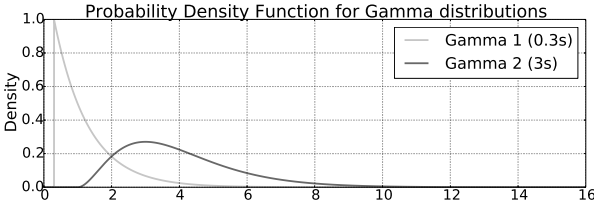


Figure 14: Propability desnity function (pdf) for the simulated *Gamma* 1 and *Gamma* 2 latency distributions.

connection types increases dramatically as we move towards slower connections like *Gamma 2*. First, AVALANCHE retrieves query specific statistics (e.g., triple pattern cardinalities and total triples) from participating endpoints. For the 0 latency setup *No Delay* this phase completes on average in 0.05 seconds and is therefore negligible compared to the overall query execution time. For the slower networks *Gamma 1* and *Gamma 2* the statistics gathering phase takes on average 1.22 seconds and 7.54 seconds respectively.

Although these execution times are significantly higher they are mainly dominated by the network connection latency when optimised remote endpoints are employed. This fact can be observed from the low response time for the same statistical information when network latency is 0. Next,

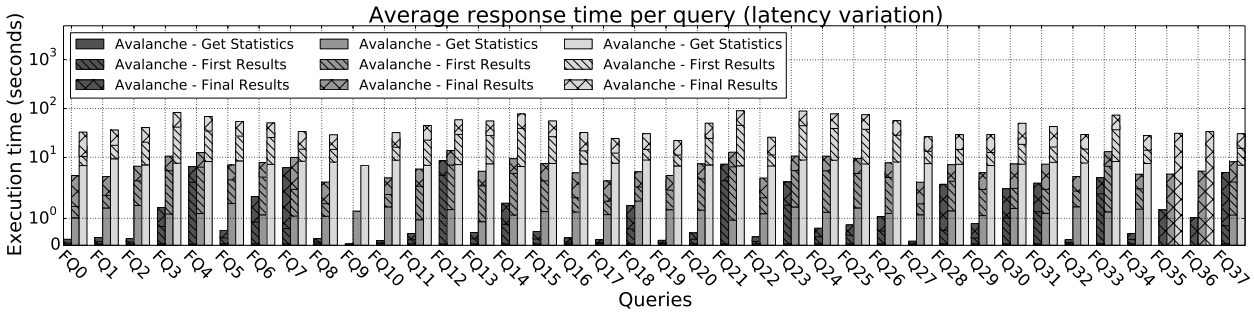


Figure 17: Average response time for each Fedbench query under different latency distributions. The graph differentiates between the time necessary to get the statistics, execute the first plan, and execute all plans.

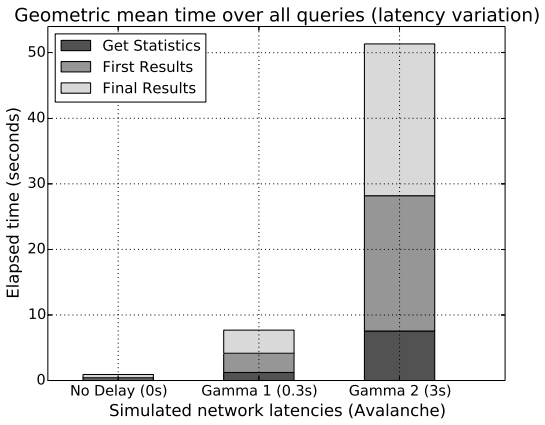


Figure 15: Geometric mean of the execution time over all queries for the three connection setups.

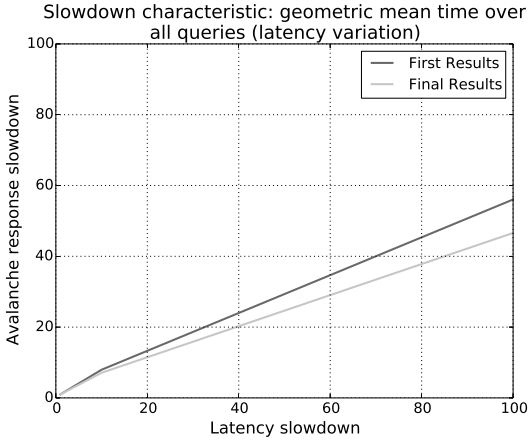


Figure 16: Slowdown introduced by the three connection setups.

results are produced after an average of 0.36 seconds when connection latency is negligible, while for the *Gamma 1* and *Gamma 2* cases first results are found after an average of 2.93 seconds and 20.64 seconds respectively. The situation is similar for achieving the stop condition or final results: 0.49 seconds on average for the *No Delay* setup, 3.52 and 23.15 seconds on average for the *Gamma 1* respectively *Gamma 2* setups. Although this performance decrease is dramatic, Avalanche exhibits a sublinear slowdown as graphed in Figure 16 compared to the broker-endpoints average latency slowdown.

This behaviour is attributed to Avalanche mainly because of its adaptive asynchronous design. In essence plans that return quickly are favoured by the asynchronous scheduling *Results Queue*. As a consequence, Avalanche is largely dependent on the *critical plan* for first results. The *critical plan* should ideally be the first productive plan. However, given that network conditions are uncontrollable, a slower plan might produce results faster because it shares a faster network connection. This is also observed in Figure 17, where the individual average times for answering all Fedbench queries $FQ_i, i \in [0, 37]$ queries under all three network conditions are graphed. As the broker-endpoints connections experience more lag, Avalanche exhibits a stable behaviour overall depending mainly on the *critical plan(s)*, albeit slower with the slowdown depicted in Figure 16.

5.1.4. Experiment #4: Varying Endpoint Availability

Another source of messiness stems from the uncontrollable nature of the underlying communication protocol stacks on the Web as well hardware and physical crashes of servers and routers. There is

no guarantee that a host replying to requests at any given moment T will be available at time $T + \Delta t$. To observe the behaviour of AVALANCHE in such a case we have designed an experiment, where some hosts disappear during query execution.

First, in order to have multiple plans per query we replicated some of the Fedbench endpoints used throughout this experimental setup. Specifically, we replicated the following AVALANCHE endpoints with a factor of 2: the *News, Movies* and *Music* in the Cross Domain collection and *Drugs* in the Life Sciences collection (see Table 2). This resulted in the increase of total number of triples over all hosts by about 8 million additional assertions. Furthermore, the already burdened physical machines had to support the 4 additional replicated endpoints.

Then, to emulate a crash the replicated endpoints were started in a "fail" mode, meaning that they would abruptly terminate themselves immediately after reporting the triple pattern cardinalities. This case is most interesting as the hosts will be considered by the *Query Planner* component as it received cardinalities from them, eventhough all query plans containing subqueries allocated to them will fail to execute. The two other cases—the host being unavailable during either the source selection or statistics gathering phase—are less interesting as they are handled by design (i.e., the hosts are not even considered in the planning). We compared AVALANCHE when replicated hosts would fail seamlessly during query execution with the case when the replicas would not fail. Note that the obtained results should not be directly compared to results obtained elsewhere in this section, as the AVALANCHE endpoints were simulated on some of the physical nodes, which experience additional load in this replicated setting.

Figure 18 graphs the arrival time of the first and total results for the cross domain and life sciences queries ($FQ_i, i \in [0, 15] \cup [33, 37]$) and Figure 19 graphs the average number of results obtained over the same queries. Note that queries $FQ9$, $FQ35$, and $FQ36$ were not considered since they produce no results be default, while query $FQ34$ could not be run in the fully replicated scenario since the physical machine did not have enough resources to accommodate the extra replicated servers in this case.

AVALANCHE's *Plan Generator* adapts dynamically to external changing conditions, such as endpoints going offline, due to various reasons. Such events are usually detected when a plan that contains at least one subquery assigned to an offline host is executed. Upon detection, the planner's internal state is dynamically readjusted first by removing the corresponding row for the host from the *Plan Matrix* P_M and secondly by pruning all partial plans containing the offline host generated up to the detection moment. In most cases AVALANCHE is not impacted by the fact that a host has failed when at least another alternate plan to produce results exists. Of course, if all query relevant hosts fail, then the query will timeout without any results found. As the results indicate AVALANCHE is able to return a result set of similar size as the one without disappearing hosts within a similar time-frame as in the stable host setting.

5.2. *Evaluation Setting II: Analyzing* AVALANCHE *with synthetic data*

One of the key characteristics of the WoD is represented by its semantic heterogeneity stemming from a plethora of intertwining applications domains. Currently this aspect alone represents an important part of a federated query's selectivity. However, it is not inconceivable that in the future *schema-homogeneous* partitions of the WoD will increase in size reducing the usefulness of schema/vocabulary information during planning. These kind of *instance-level messy* distributed RDF datasets, hence, significantly complicates distributed query processing as it is unclear if triples matching one triple pattern from one host are likely to join with matches to a second triple pattern from the same host or another. This kind of messiness attenuates the effect of locality.[25] While AVALANCHE was not designed with the intend of addressing *instance-level messiness* we investigate the behaviour of our proposed execution paradigm when individual instances (triples) are spread across a large number of *semantically-homogenous* hosts with increasing degrees of messiness.

To this end we used the synthetic LUBM benchmark dataset [13]. Specifically, we generated the LUBM2000 benchmark configuration, resulting in 2000 universities, and accounting to a total of 276 million triples. In contrast to the previous setup, where 26 **schema-heterogeneous** endpoints were

[25] Note that supporting this messiness is one underlying principles of the Semantic Web, as everyone can annotate any resource with some triple.

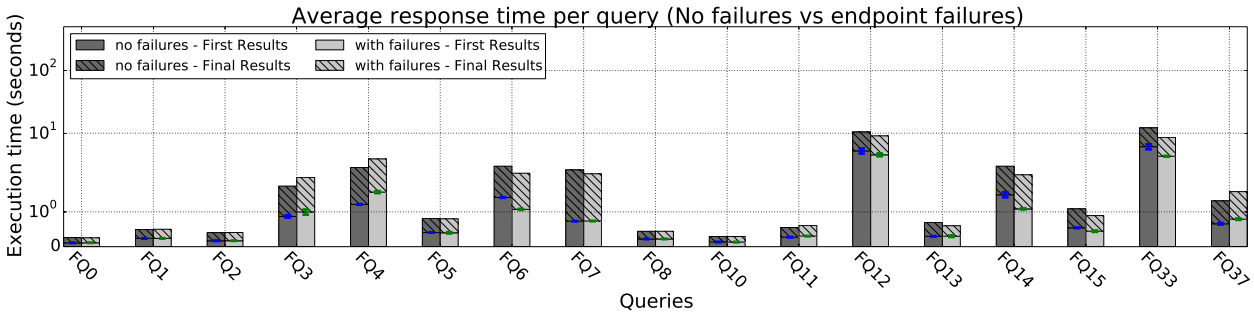


Figure 18: Average response time for Cross Domain and Life Sciences Fedbench queries when endpoints fail.

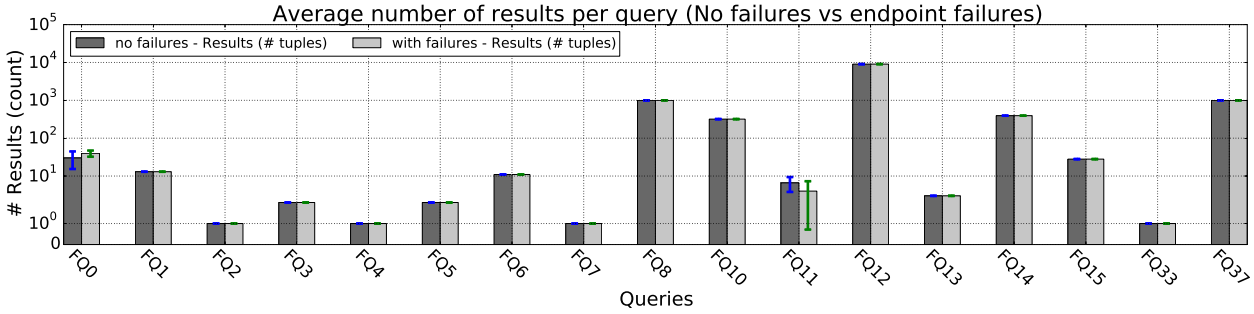


Figure 19: Average # of results time for Cross Domain and Life Sciences Fedbench queries when endpoints fail.

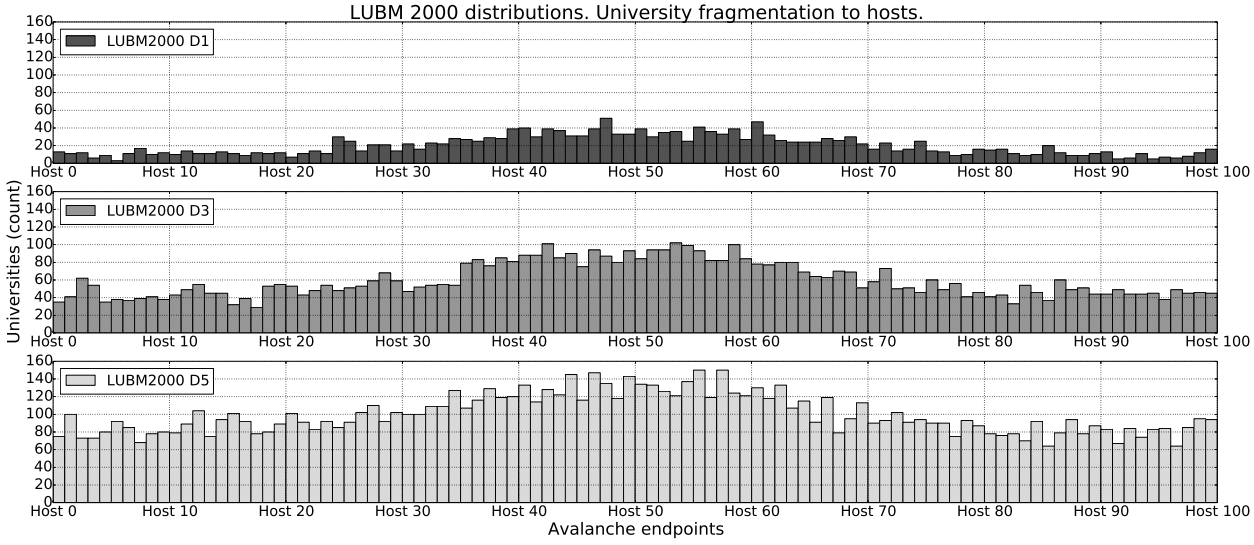


Figure 20: The data distributions chosen over 100 Hosts. The y-axis denotes the number of universities about which a host contains information.

used, a total of 100 **schema-homogeneous** endpoints are created. Such a setup allows us to flexibly mimic instance-level "distribution messiness" by reassigning triples to hosts. Note, that this setup situates AVALANCHE in a ***worst case*** scenario, where the *Source Discovery Phase* reports a large number of semantically-identical sources—all sharing the same schema—but with an unknown distribution of triples.

The Data and its distribution. As illustrated in Figure 20, the LUBM triples were allocated to hosts according to the three *LUBM2000 D1, LUBM2000 D3*, and *LUBM2000 D5* distributions (in short $D1$, $D3$ respectively $D5$). The degree of distribution messiness increases with each case as detailed in the remainder of this section.

A *coarse-grained level of messiness* is achieved in the *LUBM2000 D1* data-distribution. Here all data belonging to a university is placed on the same host. To simulate various levels of server load we assign universities to hosts using the following procedure. Half the universities are randomly assigned to a host ensuring a basic load for each host. The second half of the universities are assigned to a host by drawing the host id from a normal distribution with mean $\mu = 50$ and standard deviation $\sigma = 14$. This leads to a higher load for some hosts (towards the middle of Figure 20).

To achieve a higher degree of instance-level messiness *LUBM2000 D3 & LUBM2000 D5* additionally distribute triples of one university across 3 or even 5 hosts. The initial host for a university is still determined using the same procedure as with $D1$. Once that host is determined, however, 2 (or 4) additional hosts are randomly selected. For $D3$ each university's triples are distributed over 3 hosts using a normal distribution with $\mu = 1.5$ and $\sigma = 0.3$. similarly, for data distribution $D5$, each university's triples are distributed over 5 hosts using a normal distribution with $\mu = 2.5$ and $\sigma = 0.5$. Hence, the bulk of the university's data is still on one host with part data distributed elsewhere. This mimics a Brownian motion of the data away from its originating source – one host contains most of the data while the rest is diffused to other hosts with the chosen probability density function. Consequently, as Figure 20 shows, the hosts will have data about more universities.

The Queries. Although we employed the LUBM benchmark data generator for each of the distributions, we chose not to use the original LUBM benchmark queries since they are *a*) geared towards reasoning systems and *b*) present a coarse grain of complexity in terms of composing triple patterns and number of unbound variables rendering them unsuitable for an in-depth evaluation of AVALANCHE. Instead we devised the 11 SPARQL queries of varying complexity listed in Appendix C (listings 5 through 15) based on the observation that the number of joins involved, their size (number of participating triple patterns), and type are important descriptors of a queries' potential complexity and therefore induced effort. For example star joins can be executed in parallel as n-way joins reducing the complexity of such an operation. However, when joins are chained in a *read-after-write* manner one is forced to process them serially.

Consequently, queries $LQ_i, i \in [0, 10]$ are constructed in order of increased complexity by combining increasingly longer *read-after-write* join chains with increasingly larger sized star patterns.

5.2.1. Experiment #5: Varying Data Distribution

The results of running all eleven queries on the three data distributions ($D1$, $D3$, and $D5$) are graphed in Figures 21 and 22. All runs are warm runs and each query was run 5 times. In addition to the default values set for all experiments the following AVALANCHE stopping configuration was used: *1*) a *stop sliding window* of size 3 plans, *2*) a number of 512 maximum concurrent asynchronous connections at any given moment, and *3*) a 0.01 bloom-filter false positive error rate.

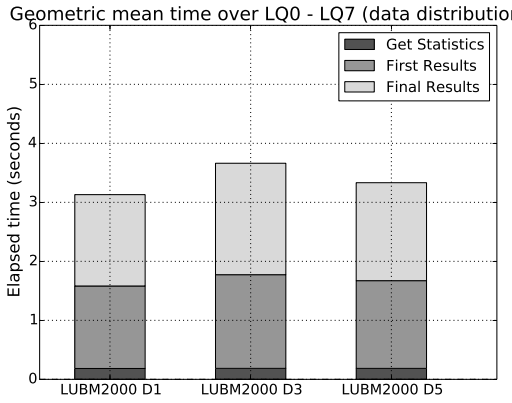


Figure 23: Geometric mean of the execution time over all queries for $D1$,$D3$ and $D5$, queries $LQ0$ through $LQ7$.

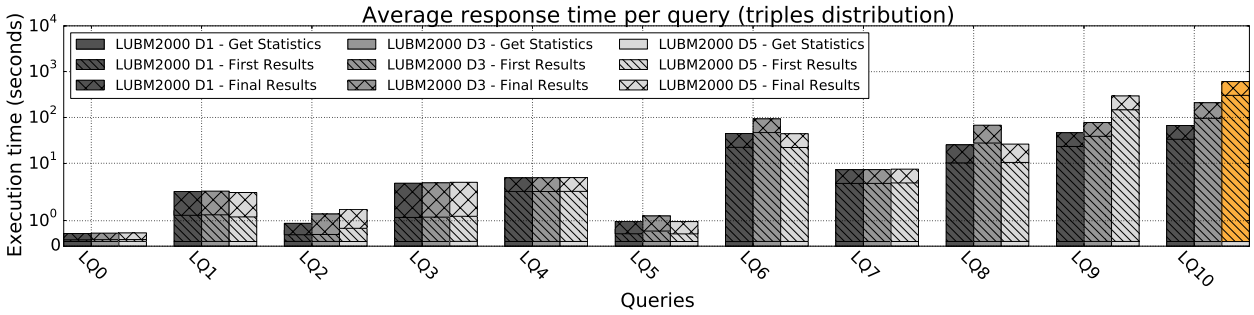


Figure 21: Query execution times for all data distributions. Timeout cases are represented with orange.

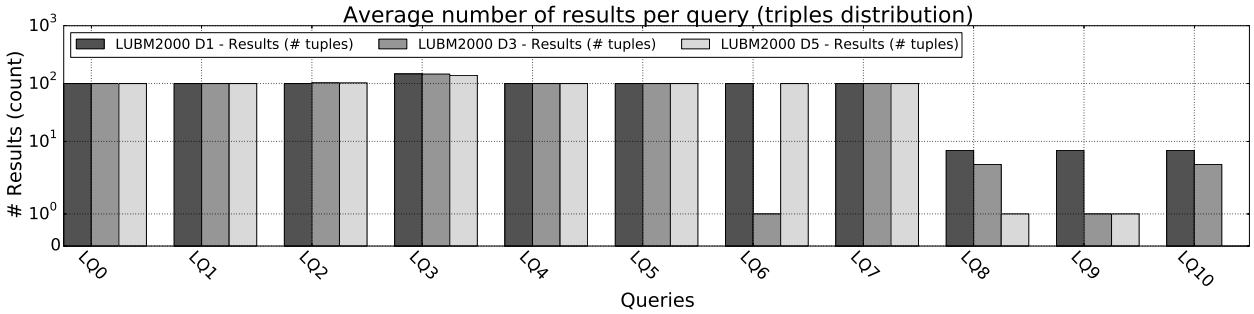


Figure 22: Number of retrieved results (average) for all data distributions.

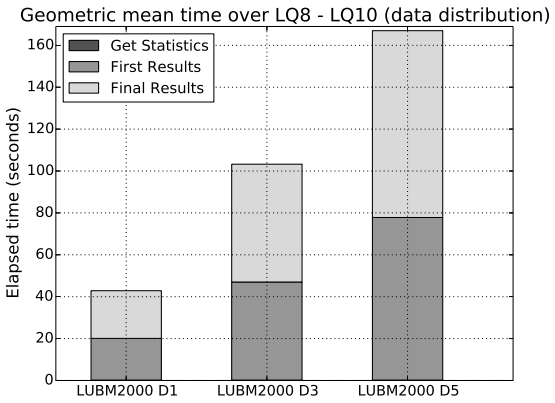


Figure 24: Geometric mean of the execution time over all queries for $D1$,$D3$ and $D5$, queries $LQ8$ through $LQ10$.

As can be observed in Figure 21 Avalanche exposes a relatively stable performance characteristic without timing-out for queries $LQ0$ through $LQ7$. Instance level spread is actually a benefiting factor for these queries that target replicated knowledge by providing more "chunks" of partial results, which in turn increases Avalanche's chances of generating a "productive" plan. Looking at Figure 22, we can clearly observe that regardless of the degree of messiness (a university's triples spread to 1, 3 or 5 endpoints), Avalanche succeeds in retrieving about the same number of results exhibiting a highly stable behavior. An exception is exhibited by $LQ6$ (Listing 11) where performance degrades only for distribution $D3$. This kind of system behavior is expected in some cases, due mainly to the estimative nature of the cost model. In this particular case the first "productive" plan is discovered relatively late compared to the other 2 distribution cases.

Queries $LQ8$, $LQ9$ and $LQ10$ form a second group of queries. These queries target very specific knowledge pertinent to a single university leaving Avalanche with the task of identifying those endpoints (1, 3 or 5), which produce the desired result when combined. As can be observed, performance

degrades dramatically with the number of hosts on which data is spread and with the number of joins generated by the query, i.e. query $FQ10$ times out (depicted in orange) for distribution $D5$ (triples spread over 5 endpoints). This result suggests that naïve selectivity estimation based cost models are not enough when dealing with *fine-grained triple-level messiness* at this scale, warranting novel and (more) accurate estimation statistics. Another effect of increased triples-spread is observed in the decline in recall for this second group of queries (Figure 22). A possible explanation for this observation is that as triples are distributed over more hosts, finding candidate joins becomes harder while the ones that are favored first are usually the more selective and, hence, the ones with fewer results.

The systems' overall behaviour for the two query groups is observed more clearly in Figures 23 and 24, where the geometric mean over answering all queries against each distribution is shown. The Figures highlight the elapsed times for the three important execution phases in Avalanche. The *statistics gathering* phase accounts for a negligible part of the entire execution process and accounts to a mere 0.2 seconds on average for both query groups. We attribute this to the *Hexastore*-inspired read optimised indexing model of the RDF store used. We observe that Avalanche exposes a stable behaviour for the first group of queries finding first answers after an average of 1.5 seconds and completing the query after an average of 1.7 seconds. For the second group of queries, Avalanche exposes a slowdown effect in terms of finding first answers, retrieving them after an average of 48 seconds and completing the query after an average of 56 seconds. Finally, while Avalanche becomes slower it however, maintains its *robustness* as it will eventually find results.

5.3. Summary

Both evaluation settings in Sections 5.1 and 5.2 are witness to Avalanche's stability against messiness. For the real world data-distribution setup based on Fedbench Avalanche was able to find first results in under one second for about 80% of the considered queries, while total results were retrieved under one second for about 70% of the queries, with the slowest running query taking about 5.5 seconds to complete. A notable exception is represented by query $FQ12$, which generates a large intermediate result set, potentially blocking or slowing down access to underlying shared resources like network connections and database indexes. This is alleviated to some extent by: first, relying on asynchronous socket API's and second, isolating the execution of expensive queries/joins inside threads or processes. Other possibilities of reducing the overhead of expensive semi-joins is by compressing intermediate result sets. Even more, a good replacement strategy for semi-joins are bloom-joins, where the actual data sent is the bit-vector forming the bloom filter of the intermediate results set. The bloom-join is advantageous for large result sets as $sizeof(ResultSet_{subquery}) >> sizeof(BitVector_{bloomfilter})$.

Furthermore, as shown in the the third experiment when the broker-endpoints network latency changes then Avalanche's slowdown compared to the connection's slowdown exhibits a sub-linear characteristic as graphed in Figure 16. Avalanche is also able to dynamically adapt when some participating endpoints go offline when they are not the sole query results providers. Considering the synthetic LUBM dataset where a "brownian" spread of triples from their source host is simulated, Avalanche exhibits a high level of stability when answering queries that are selective with respect to knowledge that is likely to be replicated (i.e. classes) as seen in Figure 23. Avalanche does become progressively slower for queries that target specific resources (Figure 24). This happens since the objective functions considered do not leverage in any way the data distribution aspect.

6. Limitations, Optimizations, and Future Work

The work presented here exhibits two kinds of limitations. On one side the system could be extended and/or optimised; on the other side the external validity of the evaluation is limited. We will discuss both of these topics in turn.

System limitations and optimisations. The Avalanche system has shown how a completely heterogeneous distributed query engine that makes no assumptions about data distribution could be implemented. The current approach does have a number of limitations as highlighted in Section 3, most notably the fact that it:

i) does not support UNION graph patterns,

ii) can be resource wasteful for some classes of query workloads, and

iii) does not offer result-set completeness guarantees.

UNIONS could be included by execution model as discussed in Section 3. One approach to address resource wastefulness would be to improve the quality of cost estimation, e.g., via learning. We intend to explore these avenues in future work. Result-set completeness is external to AVALANCHE and a characteristic of the Web-of-Data.

Furthermore, we need to better understand the cost-estimation functions used by the planner, investigate if the requirements put on participating triple-stores are reasonable, and empirically evaluate if the approach scales to an even larger number of active hosts.

To improve AVALANCHE's performance a number of research avenues and potential solutions stand out. For instance, the simplistic *source selection* algorithm can be improved with higher-quality statistics for a more accurate source selection process. Another high-impact avenue is to enhance join estimation accuracy, i.e. by using *Bloom Filters*, histograms or schema-bound join predictive models which learn join distributions from previous observations. Moreover, we intend to investigate if a stateless approach is feasible since AVALANCHE currently assumes that remote endpoints keep partial results throughout plan execution to reduce local database operational cost. Note that the simple approach—the use of REST-ful services [10]—may not be applicable as the size of the state may be too large and overburden available bandwidth. Additionally, we will need to investigate how complex it would be in practice to generalise the notion of a *common key-space* beyond the textual representation of RDF terms in order to increase the performance of bandwidth-intensive join and merge operations.

Finally, we would like to point out that AVALANCHE completely ignores schema. Whilst this allows us to provide a schema-agnostic solution it does delegate the problem to the querying user. As a large number of publications on schema-integration [9] and the owl:sameAs problem (i.e., [14]) show a lot of work might still be needed to address this kind of messiness transparently. Hence, this is beyond the scope of AVALANCHE.

Evaluation limitations. Our experiments rely on a limited number of physical resources available for accommodating the endpoints, the number of physical machines used is 4 to 16 times smaller than required in reality, where an endpoint would most often reside on an individual server. When one machine accommodates multiple endpoints, then these endpoints compete for shared resources (such as RAM, disk I/O, network I/O, and CPU-time). We think that the impact on our finding is mitigated by the choice of machines with more cores then endpoints. Furthermore, real-world endpoints would have to answers multiple query requests, each of which also competes for machine resources. Still, we believe that our setup is as realistic as possible in an experimental laboratory-setup and allows generalising the results.

In AVALANCHE we have so far focused on graph pattern matching and have thus ignored other SPARQL features like OPTIONAL and FILTER graph patterns. As part of our future work on AVALANCHE we intend to extend support to cover these features. Properly supporting FILTER graph patterns is likely to speed up query processing in AVALANCHE due to the intrinsic parallelism of union operations and due to the selective effect of filtering partial results – depending on how soon a FILTER can be evaluated.

7. Conclusion

In this paper we presented AVALANCHE, a novel approach for querying the Web-of-Data that (1) makes no assumptions about data distribution, availability, or partitioning exhibiting skew resistance for classes of queries that are selective with regards to replicated knowledge (i.e. Class information), (2) is dynamically adaptive to changing external network conditions, (3) provides up-to-date results, and (4) is flexible since it makes few limiting assumptions about the structure of participating triple stores. Specifically, we showed that AVALANCHE is able to execute non-trivial queries over distributed data-sources with an ex-ante unknown data-distribution. We showed that an extensible cost model based on a common *Multi Objective Optimisation* method—the method of *Global Criterion*, where different heuristics can be plugged in without imposing changes to existing ones—can yield good performance in spite of different data distributions or changing latency while allowing for a messy Web-of-Data.

We designed AVALANCHE with the need to handle messy semi-structured data at large scales. The core idea follows the principle of *decentralisation*.

It also supports *asynchrony* using asynchronous HTTP requests to avoid blocking, *autonomy* by delegating the coordination and execution of the distributed join/update/merge operations to the hosts, *concurrency* through the pipeline shown in Figure 3, *symmetry* by allowing each endpoint to act as the initiating AVALANCHE node for a query caller, as well as *fault tolerance* via proper exception and time-out handling and stopping conditions. By design AVALANCHE handles messiness generated by (i) schema alignment and data evolution, as AVALANCHE is schema agnostic its current view of the world is as a set of triples, (ii) data distribution through its extensible cost model, and (iii) source un-availability, as AVALANCHE dynamically dismisses plans issued to hosts that are not present anymore during the execution phase, still allowing other hosts (sources) to produce new and more results.

AVALANCHE's main limitation with respect to messiness is its assumption that participating datasources are indexed (i.e., stored in some kind of triple store rather than "just" provided as files). In the light of its robustness against other kinds of messiness, however, we believe that AVALANCHE's capabilities outweigh this disadvantage—in particular since it would be simple to "wrap" any (known)file-based source with a combination of a triple-store and crawler.

To our knowledge, AVALANCHE is the first Semantic Web query system that makes no assumptions about the data distribution whatsoever. Whilst it is only a first implementation with a number of drawbacks it represents an important step towards querying a messy Web-of-Data by embracing its messiness as necessity (rather than an impediment) in order to foster its unpredictable growth.

Acknowledgements

This work was partially supported by the Swiss National Science Foundation under contract number 200021-118000. We are also grateful to the anonymous reviewers for their constructive comments, which helped to substantially improve the paper.

References

[1] Gennady Antoshenkov and Mohamed Ziauddin. Query processing and optimization in Oracle Rdb. *The VLDB Journal The International Journal on Very Large Data Bases*, 5(4):229–237, December 1996.

[2] Medha Atre, Vineet Chaoji, Mohammed J. Zaki, and James A. Hendler. Matrix "bit" loaded: A scalable lightweight join query processor for rdf data. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 41–50, New York, NY, USA, 2010. ACM.

[3] C Basca and A Bernstein. Avalanche: Putting the spirit of the web back into semantic web querying. *The 6th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2010)*, pages 64–79, Nov 2010.

[4] Abraham Bernstein, Christoph Kiefer, and Markus Stocker. Optarq: A sparql optimization approach based on triple pattern selectivity estimation. Technical report, University of Zurich, Department of, 2007.

[5] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data - the story so far. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(3):1–22, 2009.

[6] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. Sesame: an architecure for storing and querying rdf data and schema information. *Spinning the Semantic Web*, pages 197–222, 2003.

[7] Min Cai and Martin Frank. Rdfpeers: A scalable distributed rdf repository based on a structured peer-to-peer network. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 650–657, New York, NY, USA, 2004. ACM.

[8] Orri Erling and Ivan Mikhailov. Rdf support in the virtuoso dbms. Technical report, OpenLink Software, Apr 2009.

[9] Jerome Euzenat and Pavel Shvaiko. *Ontology matching*. Springer-Verlag, Heidelberg (DE), 2007.

[10] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, May 2002.

[11] Christophe Guéret, Paul Groth, and Stefan Schlobach. erdf: Live discovery for the web of data. *Billion Triple Challenge at ISWC*, 2009.

[12] Christophe Guéret, Eyal Oren, Stefan Schlobach, and Martijn Schut. An evolutionary perspective on approximate rdf query answering. In *Proceedings of the 2Nd International Conference on Scalable Uncertainty Management*, SUM '08, pages 215–228, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semant.*, 3(2-3):158–182, October 2005.

[14] Harry Halpin, Patrick J. Hayes, James P. McCusker, Deborah L. McGuinness, and Henry S. Thompson. When owl:sameas isn't the same: An analysis of identity in linked data. *9th International Semantic Web Conference (ISWC2010)*, Nov 2010.

[15] Andreas Harth, Katja Hose, Marcel Karnstedt, Axel Polleres, Kai-Uwe Sattler, and Jürgen Umbrich. Data summaries for on-demand queries over linked data. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 411–420, New York, NY, USA, 2010. ACM.

[16] Andreas Harth, Jrgen Umbrich, Aidan Hogan, and Stefan Decker. Yars2: a federated repository for querying graph structured data from the web. *6th International Semantic Web Conference (ISWC)*, pages 211–224, 2007.

[17] Olaf Hartig, Christian Bizer, and Johann-Christoph

Freytag. Executing sparql queries over the web of linked data. In *Proceedings of the 8th International Semantic Web Conference*, ISWC '09, pages 293–309, Berlin, Heidelberg, 2009. Springer-Verlag.

[18] Olaf Hartig and Ralf Heese. The sparql query graph model for query optimization. In *Proceedings of the 4th European Conference on The Semantic Web: Research and Applications*, ESWC '07, pages 564–578, Berlin, Heidelberg, 2007. Springer-Verlag.

[19] Dennis Heimbigner and Dennis McLeod. A federated architecture for information management. *ACM Transactions on Information Systems*, 3(3):253–278, July 1985.

[20] Donald E Knuth. *The Art of Computer Programming*, volume 1, page 98. Addison-Wesley, 1997.

[21] Donald Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[22] Andreas Langegger and Wolfram Woss. Rdfstats - an extensible rdf statistics generator and library. *2012 23rd International Workshop on Database and Expert Systems Applications*, 0:79–83, 2009.

[23] Andreas Langegger, Wolfram Wöß, and Martin Blöchl. A semantic web middleware for virtual data integration on the web. In *Proceedings of the 5th European Semantic Web Conference on The Semantic Web: Research and Applications*, ESWC'08, pages 493–507, Berlin, Heidelberg, 2008. Springer-Verlag.

[24] Yingjie Li and Jeff Heflin. Using reformulation trees to optimize queries over distributed heterogeneous sources. In *Proceedings of the 9th International Semantic Web Conference on The Semantic Web - Volume Part I*, ISWC'10, pages 502–517, Berlin, Heidelberg, 2010. Springer-Verlag.

[25] Angela Maduko, Kemafor Anyanwu, Amit Sheth, and Paul Schliekelman. Estimating the cardinality of rdf graph patterns. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 1233–1234, New York, NY, USA, 2007. ACM.

[26] Kaisa Miettinen. *Nonlinear Multiobjective Optimization*, volume 12 of *International Series in Operations Research and Management Science*. Kluwer Academic Publishers, Dordrecht, 1999.

[27] Hannes Muhleisen, Tilman Walther, Anne Augustin, Marko Harasic, and Robert Tolksdorf. Configuring a self-organized semantic storage service. *The 6th International Workshop On Scalable Semantic Web Knowledge Base Systems (SSWS2010)*, pages 1–16, Nov 2010.

[28] Thomas Neumann and Gerhard Weikum. Scalable join processing on very large rdf graphs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 627–640, New York, NY, USA, 2009. ACM.

[29] Eyal Oren, Christophe Guéret, and Stefan Schlobach. Anytime query answering in rdf through evolutionary algorithms. In *International Semantic Web Conference (ISWC)*. Springer, 2008.

[30] Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems (2nd Edition)*. Prentice Hall, 2 edition, 1999.

[31] Bastian Quilitz and Ulf Leser. Querying distributed rdf data sources with sparql. In *Proceedings of the 5th European Semantic Web Conference on The Semantic Web: Research and Applications*, ESWC'08, pages 524–538, Berlin, Heidelberg, 2008. Springer-Verlag.

[32] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[33] Sukriti Ramesh, Odysseas Papapetrou, and Wolf Siberski. Optimizing distributed joins with bloom filters. In Manish Parashar and Sanjeev K. Aggarwal, editors, *Distributed Computing and Internet Technology*, volume 5375 of *Lecture Notes in Computer Science*, pages 145–156. Springer Berlin Heidelberg, 2009.

[34] Simon Schenk and Steffen Staab. Networked graphs: A declarative mechanism for sparql rules, sparql views and rdf data integration on the web. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 585–594, New York, NY, USA, 2008. ACM.

[35] Michael Schmidt, Olaf G orlitz, Peter Haase, G unter Ladwig, Andreas Schwarte, and Thanh Tran. Fedbench: A benchmark suite for federated semantic data query processing. In Lora Aroyo, Chris Welty, Harith Alani, Jamie Taylor, Abraham Bernstein, Lalana Kagal, Natasha Noy, and Eva Blomqvist, editors, *The Semantic Web ISWC 2011*, volume 7031 of *Lecture Notes in Computer Science*, pages 585–600. Springer Berlin Heidelberg, 2011.

[36] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.*, 22(3):183–236, September 1990.

[37] Heiner Stuckenschmidt, Richard Vdovjak, Geert-Jan Houben, and Jeen Broekstra. Index structures and algorithms for querying distributed rdf repositories. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 631–639, New York, NY, USA, 2004. ACM.

[38] Giovanni Tummarello, Richard Cyganiak, Michele Catasta, Szymon Danielczyk, Renaud Delbru, and Stefan Decker. Sig.ma: Live views on the Web of Data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 8(4):355–364, November 2010.

[39] Octavian Udrea, College Park, and Andrea Pugliese. GRIN : A Graph Based RDF Index. *Artificial Intelligence*, 22(2):1465–1470, 2007.

[40] Marshall Van Alstyne, Erik Brynjolfsson, and Stuart Madnick. Why not one big database? principles for data ownership. *Decis. Support Syst.*, 15(4):267–284, December 1995.

[41] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: Sextuple indexing for semantic web data management. *Proc. VLDB Endow.*, 1(1):1008–1019, August 2008.

[42] Milan Zeleny. Compromise programming. In Cochrane James L. and Milan Zeleny, editors, *Multiple Criteria Decision Making*, pages 262–301. University of South Carolina Press, Columbia, 1973.

[43] Jan Zemanek, Simon Schenk, and Vojtech Svatek. Optimizing sparql queries over disparate rdf data sources through distributed semi-joins. In *ISWC 2008 Poster and Demo Session Proceedings*. CEUR-WS, 2008.

Appendix A. Avalanche Endpoint Operators

Execution Operators. For brevity, example query listings will not include the prefixes already defined in the motivating example query Q_{ex}.

`getTPCardinality(`*tp*`)`

As the name suggests, this operator is responsible with returning the number of instances matching the triple pattern *tp* on the callee endpoint. This operator is SPARQL (1.1) compliant and can be implemented in several fashions depending on wether the predicate is bound and VoID is used. To illustrate how, the following triple pattern from Q_{ex} is considered:
`< ?chebiDrug, chebi:image, ?chebiImage >`.

`Example: getTPCardinality operator to SPARQL(1.1) mapping`

```
PREFIX void: <http://rdfs.org/ns/void#>

## If predicate is bound and VoID is used
SELECT ?cardinality WHERE {
 ?dataset void:propertyPartition
      ?partition .
 ?partition void:property chebi:image .
 ?partition void:triples ?cardinality
}

## If VoID is not used but SPARQL 1.1
     compliant
SELECT (COUNT(DISTINCT ?chebiDrug) as
     ?cardinality) WHERE {
 ?chebiDrug chebi:image ?chebiImage
}
```

`getTotalTriples()`

SPARQL compliant as well, this is arguably the simplest operator. Its task being to report the total number of triples indexed by the endpoint. The overwhelming majority of modern day triple stores are aware of this fact and exposing this as a VoID statistic would be trivial.

`Example: getTotalTriples operator to SPARQLmapping`

```
## if VoID is used
SELECT ?dataset ?total WHERE {
 ?dataset void:triples ?total .
}
```

`executeQuery(`*bgp*`, `*vars*`, `*values*`)`

This operator is virtually implemented by all RDF triple stores. The optional *vars* and *values* arguments are mapped directly to the `VALUES` term in SPARQL 1.1. For example consider the second fragment from Q_{ex} in Listing 2 with example dummy *values* for the `?drugBankName` variable:

`Example: executeQuery operator to SPARQL1.1 mapping`

```
SELECT ?chebiDrug ?chebiImage WHERE {
 ?chebiDrug chebi:image ?chebiImage .
 ?chebiDrug dc:title ?drugBankName
} VALUES (?drugBankName) {
 ("Drug A")
 ("Drug B")
 ("Drug C")
}
```

`executeDistributedJoin(`bgp_{local}`, `bgp_{remote}`, `*host*`)`

A critical part of the core functionality of any distributed database querying system is given by the ability to execute distributed joins. This operator is essentially a *proxy* operator as it relies on the ability to execute SPARQL queries both locally and remotely and functions as following: first the subquery bgp_{local} is executed locally as any regular SPARQL query. Next, the join variables (*vars*) between the two subqueries (bgp_{local} and bgp_{remote}) are determined and the partial results corresponding to them are selected (*values*). As the final step the `executeQuery(`bgp_{remote}`, `*vars*`, `*values*`)` operator is called on the remote *host*.

The following operator pair is optional and exists mainly for optimization reasons. Their role is to simply reduce the end I/O cost of executing a distributed query:

`executeDistributedReconciliation(`bgp_{local}`, `bgp_{remote}`, `*host*`)`

Regarded as a "cleanup" operation the set-reconciliation procedure follows the execution of a distributed n-way join in order to remove partial results in excess resulting from preceding joins. Also a *proxy* operator baring a simplistic nature, its task is that of determining the *values* of the join *vars* between the two subqueries (bgp_{local} and bgp_{remote}) and calling `executeReconciliation(`bgp_{remote}`, `*vars*`, `*values*`)` on the remote *host*. Various optimizations are possible at this stage. Hence, instead of sending the actual set of *values* (compressed or not), a set of their hashes or a **bloom filter** can be employed, resulting in a hash- or a bloom filter-optimized distributed join.

`executeReconciliation(`*bgp*`, `*vars*`, `*values*`)`

Always called as the result of executing the `executeDistributedReconciliation` operator, its scope is to select and filter the excess results corresponding to the previously locally executed *bgp* query. As mentioned earlier this operator is designed to reduce the network traffic for the final *merge* phase of the distributed query execution. Depending on the optimization mechanism used (hashing, bloom filters, or the actual set) the process can be exact or exhibit false positives (for bloom filters).

The following operators are required in the final stages of the query execution process:

`executeDistributedMerge(`bgp_{local}`, `bgp_{remote}`, `*host*`)`

Just like the previous `executeDistributedJoin` operator, this is also a proxy operator paired with `executeMerge`. The partial results contained in *results_table* corresponding to the previously executed query bgp_{local} are selected and sent remotely by calling `executeMerge(`bgp_{remote}`, `*results_table*`)` on *host*. This operator is outside the scope of SPARQL compliancy, however, it can be implemented as a simple HTTP GET call as described by the REST model [10].

`executeMerge(bgp, results_table)`

Called as a result of a distributed merge operation, this final operator in the execution pipeline implements the standard database INNER JOIN (⋈) operation on the incoming remote *results_table* and the local partial results table corresponding to the *bgp* query, which was previously executed during the distributed join phase.

`materialize(bgp)`

This operator is necessary when distributed joins are executed in a common ID space used by the remote endpoints to index RDF data-sets. As the name suggests its basic functionality is that of providing the mapping from ID to RDF literals, a necessary condition when formulating the final results.

State Management Operators. The following state management operators[26] are exposed by AVALANCHE as a means to allow query brokers to halt the distributed operations involved in answering a query when the desired results are found:

`stopPlan(pid)`

Although not strictly necessary for AVALANCHE to function, the operator ensures the "cleanup" and freeing of allocated resources while trying to satisfy a given plan denoted by the *pid* identifier (i.e. the MD5 hash of the SPARQL 1.1 query decomposition).

`stopAllPlans(Q)`

Similarly, the operator will stop the execution and free all resources allocated for the resolving of all plans pertaining to the considered query. To reduce network overhead the query string can be replaced with a simple hash of the actual query (i.e., the MD5 hash of the original SPARQL query).

Appendix B. Fedbench Query Name Mapping

Appendix C. LUBM Benchmark Queries

```
PREFIX lubm: <http://www.lehigh.edu/~zhp2/2004/0401/
    univ-bench.owl#>
PREFIX uni0: <http://www.Department1.University0.edu/>
```

Listing 4: PREFIXES

```
SELECT ?professor WHERE {
 ?professor lubm:name "FullProfessor1"} LIMIT 100
```

Listing 5: LQ0

```
SELECT ?department ?researchGroups WHERE {
 ?researchGroups lubm:subOrganizationOf ?department.
 ?department lubm:name "Department1"} LIMIT 100
```

Listing 6: LQ1

[26]Both operators can be implemented as REST calls

```
SELECT ?studentName WHERE {
 ?student lubm:name ?studentName.
 ?student lubm:memberOf <http://www.Department1.
      University0.edu>} LIMIT 100
```

Listing 7: LQ2

```
SELECT ?property ?value WHERE {
 ?professor lubm:name "FullProfessor1".
 ?professor ?property ?value} LIMIT 100
```

Listing 8: LQ3

```
SELECT ?mail ?phone WHERE {
 ?professor lubm:emailAddress ?mail.
 ?professor lubm:telephone ?phone.
 ?professor lubm:name "FullProfessor1"} LIMIT 100
```

Listing 9: LQ4

```
SELECT ?mail ?phone ?doctor WHERE {
 ?professor lubm:emailAddress ?mail.
 ?professor lubm:telephone ?phone.
 ?professor lubm:doctoralDegreeFrom ?doctor.
 ?professor lubm:name "FullProfessor1"} LIMIT 100
```

Listing 10: LQ5

```
SELECT ?studentName ?courseName WHERE {
 ?student lubm:takesCourse ?course.
 ?course lubm:name ?courseName.
 ?student lubm:name ?studentName.
 ?student lubm:memberOf <http://www.Department1.
      University0.edu>} LIMIT 100
```

Listing 11: LQ6

```
SELECT ?publication ?author ?department ?university
    WHERE {
 ?publication lubm:name "Publication0".
 ?publication lubm:publicationAuthor ?author.
 ?author lubm:worksFor ?department.
 ?department lubm:subOrganizationOf ?university} LIMIT
      100
```

Listing 12: LQ7

```
SELECT ?name ?advisor ?department WHERE {
 ?advisor lubm:worksFor ?department.
 ?student lubm:advisor ?advisor.
 ?student lubm:name ?name.
 ?student lubm:takesCourse uni0:GraduateCourse33}
      LIMIT 100
```

Listing 13: LQ8

```
SELECT ?name ?tel ?advisor ?department WHERE {
 ?advisor lubm:worksFor ?department.
 ?student lubm:advisor ?advisor.
 ?student lubm:name ?name.
 ?student lubm:telephone ?tel.
 ?student lubm:takesCourse uni0:GraduateCourse33}
      LIMIT 100
```

Listing 14: LQ9

```
SELECT ?university ?student ?name ?tel WHERE {
 ?student lubm:advisor ?advisor.
 ?advisor lubm:worksFor ?department.
 ?department lubm:subOrganizationOf ?university.
 ?student lubm:name ?name.
 ?student lubm:telephone ?tel.
 ?student lubm:takesCourse uni0:GraduateCourse33}
      LIMIT 100
```

Listing 15: LQ10

Table B.5: Fedbench query name mapping

Collection	Fedbench Name	Name	Fedbench Name	Name	Fedbench Name	Name
Cross Domain	CD 1a[c]	**FQ0**	CD 1b[c]	**FQ1**	CD 2	**FQ2**
	CD 3	**FQ3**	CD 4	**FQ4**	CD 5	**FQ5**
	CD 6	**FQ6**	CD 7	**FQ7**		
Life Sciences	LS 1a[c]	**FQ8**	LS 1b[c]	**FQ9**	LS 2a[c]	**FQ10**
	LS 2b[c]	**FQ11**	LS 3	**FQ12**	LS 4	**FQ13**
	LS 5	**FQ14**	LS 6	**FQ15**		
Life Sciences + [b]		**FQ33**		**FQ34**		**FQ35**
		FQ36		**FQ37**		
Linked Data	LD 1	**FQ16**	LD 2	**FQ17**	LD 3	**FQ18**
	LD 4	**FQ19**	LD 5	**FQ20**	LD 6	**FQ21**
	LD 7	**FQ22**	LD 8	**FQ23**	LD 9	**FQ24**
	LD 10	**FQ25**	LD 11	**FQ26**		
SP^2Bench	SP2Bench Q1	**FQ27**	SP2Bench Q2[d]	**FQ28**	SP2Bench Q5	**FQ29**
	SP2Bench Q9a[c]	**FQ30**	SP2Bench Q9b[c]	**FQ31**	SP2Bench Q10	**FQ32**

[a] Original query names from the Fedbench project: http://code.google.com/p/fbench/wiki/Queries.
[b] These queries are not part of the original Fedbench benchmark and therefore do not have a corresponding denomination. They are added for their increased complexity.
[c] Queries whose names are suffixed with *a* or *b* represent Fedbench queries that contain UNION graph patterns. The two subqueries are executed independently.
[d] Since the version of Avalanche used for this evaluation does not support the OPTIONAL graph pattern modifier, any OPTIONAL graph patterns were discarded from the query.

Appendix D. Fedbench Benchmark Queries

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX kegg: <http://bio2rdf.org/ns/kegg#>
PREFIX nytimes: <http://data.nytimes.com/elements/>
PREFIX geonames: <http://www.geonames.org/ontology#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX swc: <http://data.semanticweb.org/ns/swc/
    ontology#>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX bench: <http://localhost/vocabulary/bench/>
PREFIX drugbank: <http://www4.wiwiss.fu-berlin.de/
    drugbank/resource/drugbank/>
PREFIX person: <http://localhost/persons/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-
    ns#>
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX swrc: <http://swrc.ontoware.org/ontology#>
PREFIX drugbank-category: <http://www4.wiwiss.fu-
    berlin.de/drugbank/resource/drugcategory/>
PREFIX drugbank-drugs: <http://www4.wiwiss.fu-berlin.
    de/drugbank/resource/drugs/>
PREFIX linkedmdb: <http://data.linkedmdb.org/resource/
    movie/>
PREFIX chebi: <http://bio2rdf.org/ns/bio2rdf#>
PREFIX purl: <http://purl.org/dc/terms/>
```

Listing 16: PREFIXES

```
SELECT ?predicate ?object WHERE {
 dbpedia:Barack_Obama ?predicate ?object}
```

Listing 17: FQ0

```
SELECT ?predicate ?object WHERE {
 ?subject owl:sameAs dbpedia:Barack_Obama.
 ?subject ?predicate ?object}
```

Listing 18: FQ1

```
SELECT ?party ?page WHERE {
 dbpedia:Barack_Obama dbpedia-owl:party ?party.
 ?x nytimes:topicPage ?page.
 ?x owl:sameAs dbpedia:Barack_Obama}
```

Listing 19: FQ2

```
SELECT ?president ?party ?x WHERE {
 ?president rdf:type dbpedia-owl:President.
 ?president dbpedia-owl:nationality dbpedia:
     United_States.
 ?president dbpedia-owl:party ?party.
 ?x nytimes:topicPage ?page.
 ?x owl:sameAs ?president}
```

Listing 20: FQ3

```
SELECT ?actor ?news WHERE {
 ?film purl:title "Tarzan".
 ?film linkedmdb:actor ?actor.
 ?actor owl:sameAs ?x.
 ?y owl:sameAs ?x.
 ?y nytimes:topicPage ?news}
```

Listing 21: FQ4

```
SELECT ?film ?director ?genre WHERE {
 ?film dbpedia-owl:director ?director.
 ?director dbpedia-owl:nationality dbpedia:Italy.
 ?x owl:sameAs ?film.
 ?x linkedmdb:genre ?genre}
```

Listing 22: FQ5

```
SELECT ?name ?location WHERE {
 ?artist foaf:name ?name.
 ?artist foaf:based_near ?location.
 ?location geonames:parentFeature ?germany.
 ?germany geonames:name "Federal Republic of Germany"}
```

Listing 23: FQ6

```
SELECT ?location ?news WHERE {
 ?location geonames:parentFeature ?parent.
 ?parent geonames:name "California".
 ?y owl:sameAs ?location.
 ?y nytimes:topicPage ?news}
```

Listing 24: FQ7

```
SELECT ?drug ?melt WHERE {
 ?drug drugbank:meltingPoint ?melt}
```

Listing 25: FQ8

```
SELECT ?drug ?melt WHERE {
 ?drug dbpedia-owl:drug/meltingPoint ?melt}
```

Listing 26: FQ9

```
SELECT ?predicate ?object WHERE {
 drugbank-drugs:DB00201 ?predicate ?object}
```

Listing 27: FQ10

```
SELECT ?predicate ?object WHERE {
 drugbank-drugs:DB00201 owl:sameAs ?caff.
 ?caff ?predicate ?object}
```

Listing 28: FQ11

```
SELECT ?Drug ?IntDrug ?IntEffect WHERE {
 ?Drug rdf:type dbpedia-owl:Drug.
 ?y owl:sameAs ?Drug.
 ?Int drugbank:interactionDrug1 ?y.
 ?Int drugbank:interactionDrug2 ?IntDrug.
 ?Int drugbank:text ?IntEffect}
```

Listing 29: FQ12

```
SELECT ?drugDesc ?cpd ?equation WHERE {
 ?drug drugbank:drugCategory drugbank-category:
      cathartics.
 ?drug drugbank:keggCompoundId ?cpd.
 ?drug drugbank:description ?drugDesc.
 ?enzyme kegg:xSubstrate ?cpd.
 ?enzyme rdf:type kegg:Enzyme.
 ?reaction kegg:xEnzyme ?enzyme.
 ?reaction kegg:equation ?equation}
```

Listing 30: FQ13

```
SELECT ?drug ?keggUrl ?chebiImage WHERE {
 ?drug rdf:type drugbank:drugs.
 ?drug drugbank:keggCompoundId ?keggDrug.
 ?keggDrug chebi:url ?keggUrl.
 ?drug drugbank:genericName ?drugBankName.
 ?chebiDrug dc:title ?drugBankName.
 ?chebiDrug chebi:image ?chebiImage}
```

Listing 31: FQ14

```
SELECT ?drug ?title WHERE {
 ?drug drugbank:drugCategory drugbank-category:
      micronutrient.
 ?drug drugbank:casRegistryNumber ?id.
 ?keggDrug rdf:type kegg:Drug.
 ?keggDrug chebi:xRef ?id.
 ?keggDrug dc:title ?title}
```

Listing 32: FQ15

```
SELECT ?paper ?p ?n WHERE {
 ?paper swc:isPartOf <http://data.semanticweb.org/
      conference/iswc/2008/poster_demo_proceedings>.
 ?paper swrc:author ?p.
 ?p rdfs:label ?n}
```

Listing 33: FQ16

```
SELECT ?proceedings ?paper ?p WHERE {
 ?proceedings swc:relatedToEvent <http://data.
      semanticweb.org/conference/eswc/2010>.
 ?paper swc:isPartOf ?proceedings.
 ?paper swrc:author ?p}
```

Listing 34: FQ17

```
SELECT ?paper ?p ?x ?n WHERE {
 ?paper swc:isPartOf <http://data.semanticweb.org/
      conference/iswc/2008/poster_demo_proceedings>.
 ?paper swrc:author ?p.
 ?p owl:sameAs ?x.
 ?p rdfs:label ?n}
```

Listing 35: FQ18

```
SELECT ?role ?p ?paper ?proceedings WHERE {
 ?role swc:isRoleAt <http://data.semanticweb.org/
      conference/eswc/2010>.
 ?role swc:heldBy ?p.
 ?paper swrc:author ?p.
 ?paper swc:isPartOf ?proceedings.
 ?proceedings swc:relatedToEvent <http://data.
      semanticweb.org/conference/eswc/2010>}
```

Listing 36: FQ19

```
SELECT ?a ?n WHERE {
 ?a dbpedia-owl:artist dbpedia:Michael_Jackson.
 ?a rdf:type dbpedia-owl:Album.
 ?a foaf:name ?n}
```

Listing 37: FQ20

```
SELECT ?director ?film ?x ?y ?n WHERE {
 ?director dbpedia-owl:nationality dbpedia:Italy.
 ?film dbpedia-owl:director ?director.
 ?x owl:sameAs ?film.
 ?x foaf:based_near ?y.
 ?y geonames:officialName ?n}
```

Listing 38: FQ21

```
SELECT ?x ?n WHERE {
 ?x geonames:parentFeature <http://sws.geonames.org
      /2921044/>.
 ?x geonames:name ?n}
```

Listing 39: FQ22

```
SELECT ?drug ?id ?s ?o ?sub WHERE {
 ?drug drugbank:drugCategory drugbank-category:
      micronutrient.
 ?drug drugbank:casRegistryNumber ?id.
 ?drug owl:sameAs ?s.
 ?s foaf:name ?o.
 ?s skos:subject ?sub}
```

Listing 40: FQ23

```
SELECT ?x ?p WHERE {
 ?x skos:subject dbpedia:Category:FIFA_World_Cup-
      winning_countries.
 ?p dbpedia-owl:managerClub ?x.
 ?p foaf:name "Luiz Felipe Scolari"}
```

Listing 41: FQ24

```
SELECT ?n ?p2 ?u WHERE {
 ?n skos:subject dbpedia:Category:
      Chancellors_of_Germany.
 ?n owl:sameAs ?p2.
 ?p2 nytimes:latest_use ?u}
```

Listing 42: FQ25

```
SELECT ?x ?y ?d ?p ?l WHERE {
 ?x dbpedia-owl:team dbpedia:Eintracht_Frankfurt.
 ?x rdfs:label ?y.
 ?x dbpedia-owl:birthDate ?d.
 ?x dbpedia-owl:birthPlace ?p.
 ?p rdfs:label ?l}
```

Listing 43: FQ26

```
SELECT ?yr WHERE {
 ?journal rdf:type bench:Journal.
 ?journal dc:title "Journal 1 (1940)".
 ?journal purl:issued ?yr}
```

Listing 44: FQ27

```
SELECT ?inproc ?author ?booktitle ?title ?proc ?ee ?
     page ?url ?yr WHERE {
 ?inproc rdf:type bench:Inproceedings.
 ?inproc dc:creator ?author.
 ?inproc bench:booktitle ?booktitle.
 ?inproc dc:title ?title.
 ?inproc purl:partOf ?proc.
 ?inproc rdfs:seeAlso ?ee.
 ?inproc swrc:pages ?page.
 ?inproc foaf:homepage ?url.
 ?inproc purl:issued ?yr}
```

Listing 45: FQ28

```
SELECT ?person ?name WHERE {
 ?article rdf:type bench:Article.
 ?article dc:creator ?person.
 ?inproc rdf:type bench:Inproceedings.
 ?inproc dc:creator ?person.
 ?person foaf:name ?name}
```

Listing 46: FQ29

```
SELECT ?predicate WHERE {
 ?person rdf:type foaf:Person.
 ?subject ?predicate ?person}
```

Listing 47: FQ30

```
SELECT ?predicate WHERE {
 ?person rdf:type foaf:Person.
 ?person ?predicate ?object}
```

Listing 48: FQ31

```
SELECT ?subject ?predicate WHERE {
 ?subject ?predicate person:Paul_Erdoes}
```

Listing 49: FQ32

```
SELECT ?drug ?enzyme ?reaction WHERE {
 ?drug1 drugbank:drugCategory drugbank-category:
      antibiotics.
 ?drug2 drugbank:drugCategory drugbank-category:
      antiviralAgents.
 ?drug3 drugbank:drugCategory drugbank-category:
      antihypertensiveAgents.
 ?I1 drugbank:interactionDrug2 ?drug1.
 ?I1 drugbank:interactionDrug1 ?drug.
 ?I2 drugbank:interactionDrug2 ?drug2.
 ?I2 drugbank:interactionDrug1 ?drug.
 ?I3 drugbank:interactionDrug2 ?drug3.
 ?I3 drugbank:interactionDrug1 ?drug.
 ?drug owl:sameAs ?drug5.
 ?drug5 rdf:type dbpedia-owl:Drug.
 ?drug drugbank:keggCompoundId ?cpd.
 ?enzyme kegg:xSubstrate ?cpd.
 ?enzyme rdf:type kegg:Enzyme.
 ?reaction kegg:xEnzyme ?enzyme.
 ?reaction kegg:equation ?equation}
```

Listing 50: FQ33

```
SELECT ?drug ?drug1 ?drug2 ?drug3 ?drug4 WHERE {
 ?drug1 drugbank:drugCategory drugbank-category:
      antibiotics.
 ?drug2 drugbank:drugCategory drugbank-category:
      antiviralAgents.
 ?drug3 drugbank:drugCategory drugbank-category:
      antihypertensiveAgents.
 ?drug4 drugbank:drugCategory drugbank-category:anti-
      bacterialAgents.
 ?I1 drugbank:interactionDrug2 ?drug1.
 ?I1 drugbank:interactionDrug1 ?drug.
 ?I2 drugbank:interactionDrug2 ?drug2.
 ?I2 drugbank:interactionDrug1 ?drug.
 ?I3 drugbank:interactionDrug2 ?drug3.
 ?I3 drugbank:interactionDrug1 ?drug.
 ?I4 drugbank:interactionDrug2 ?drug4.
 ?I4 drugbank:interactionDrug1 ?drug}
```

Listing 51: FQ34

```
SELECT ?drug WHERE {
 ?drug1 drugbank:possibleDiseaseTarget <http://www4.
      wiwiss.fu-berlin.de/diseasome/resource/diseases
      /302>.
 ?drug2 drugbank:possibleDiseaseTarget <http://www4.
      wiwiss.fu-berlin.de/diseasome/resource/diseases
      /53>.
 ?drug3 drugbank:possibleDiseaseTarget <http://www4.
      wiwiss.fu-berlin.de/diseasome/resource/diseases
      /59>.
 ?drug4 drugbank:possibleDiseaseTarget <http://www4.
      wiwiss.fu-berlin.de/diseasome/resource/diseases
      /105>.
 ?I1 drugbank:interactionDrug2 ?drug1.
 ?I1 drugbank:interactionDrug1 ?drug.
 ?I2 drugbank:interactionDrug2 ?drug2.
 ?I2 drugbank:interactionDrug1 ?drug.
 ?I3 drugbank:interactionDrug2 ?drug3.
 ?I3 drugbank:interactionDrug1 ?drug.
 ?I4 drugbank:interactionDrug2 ?drug4.
 ?I4 drugbank:interactionDrug1 ?drug.
 ?drug drugbank:casRegistryNumber ?id.
 ?keggDrug rdf:type kegg:Drug.
 ?keggDrug chebi:xRef ?id.
 ?keggDrug dc:title ?title}
```

Listing 52: FQ35

```
SELECT ?d ?drug5 ?cpd ?enzyme ?equation WHERE {
 ?drug1 drugbank:possibleDiseaseTarget <http://www4.
      wiwiss.fu-berlin.de/diseasome/resource/diseases
      /261>.
 ?I1 drugbank:interactionDrug2 ?drug1.
 ?I1 drugbank:interactionDrug1 ?drug.
 ?drug drugbank:possibleDiseaseTarget ?d.
 ?drug owl:sameAs ?drug5.
 ?drug5 rdf:type dbpedia-owl:Drug.
 ?drug drugbank:keggCompoundId ?cpd.
 ?enzyme kegg:xSubstrate ?cpd.
 ?enzyme rdf:type kegg:Enzyme.
 ?reaction kegg:xEnzyme ?enzyme.
 ?reaction kegg:equation ?equation}
```

Listing 53: FQ36

```
SELECT ?drug5 ?drug6 WHERE {
 ?drug1 drugbank:possibleDiseaseTarget <http://www4.
      wiwiss.fu-berlin.de/diseasome/resource/diseases
      /319>.
 ?drug1 drugbank:possibleDiseaseTarget <http://www4.
      wiwiss.fu-berlin.de/diseasome/resource/diseases
      /270>.
 ?I1 drugbank:interactionDrug1 ?drug1.
 ?I1 drugbank:interactionDrug2 ?drug.
 ?drug1 owl:sameAs ?drug5.
 ?drug owl:sameAs ?drug6}
```

Listing 54: FQ37